Chapitre 1

Chaînes de caractères

1.1 détecter tous les anagrames

```
entrée:  soir loup isor lune dent
sortie:  {soir,isor}
```

Un mot w est un anagrame d'un mot v, s'il existe une permutation des lettres qui transforme w en v.

Étant donnée un ensemble de n mots de longueur au plus m, on veut détecter tous les anagrames.

Notre solution résoud ce problème en temps $O(nm \log m)$. L'idée est de calculer une signature pour chaque mot, tel que deux mots sont anagrames l'un de l'autre si et seulement s'ils ont la même signature. Cette signature est tout simplement un autre mot composé des mêmes lettres, mais ordonnées.

Notre structure de donnée est un dictionnaire qui associe à chaque signature la liste des mots qui ont cette signature.

```
def anagrames(w):
    # -- grouper les mots par même signature
    d = {}
    for i in range(len(w)):
        s = ''.join(sorted(w[i])) # -- signature
        if s in d:
          d[s].append(i)
        else:
          d[s] = [i]
    # -- extraire anagrames
    for s in d:
        if len(d[s])>1: # ignorer mots sans anagrame
          for i in d[s]:
              print w[i],
          print
```

Problèmes Anagram Groups [poj :2408], T9 [poj :1451]

Chapitre 2

Séquences

2.1 Plus long palindrome

```
entrée:  babcbabcbaccba
sortie:   abcbabcba
```

Un mot s est un palindrome si le premier caractère de s est égal au dernier, le 2ème est égal à l'avant-dernier et ainsi de suite.

Le problème du plus long palindrome consiste à déterminer la plus longue sous-chaîne qui soit un palindrome. Ce problème peut-être résolu en temps quadratique par l'algorithme naïf, en temps $O(n \log n)$ avec des tableaux de suffixes et en temps $O(n)$ par l'algorithme de Manacher que nous décrivons ici.

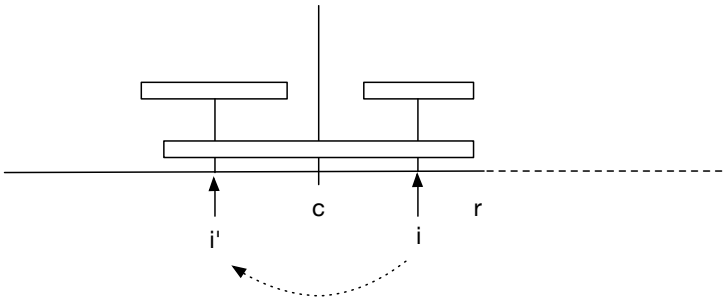


FIGURE 2.1 – L'algorithme de Manacher. On a déjà calculé p pour les indices $< i$, et on veut calculer $p[i]$. Soit un palindrome centré en c de rayon $r - c$ avec r maximal, et i' le mirroir de i par rapport à c. Par symmétrie, le palindrome centré en i' de rayon $p[i']$ doit être égal au mot centré en i, au moins jusqu'au rayon $r - i$.

D'abord on transforme l'entrée s en insérant un séparateur '#' autour de chaque caractère. Par exemple `abc` se transforme en `#a#b#c`. Soit t le résultat. Ceci permet de traiter de manière égale les palindromes de longueur pair et de longueur impair. Notez qu'avec cette transformation tout palindrome commence et termine par un séparateur '#'. Ainsi les extrémités d'un palindrome ont des indices de même parité, et ceci simplifie alors la transformation d'une solution sur la chaîne t en une solution sur la chaîne s.

La sortie de l'algorithme est un tableau p qui indique pour chaque position i, le plus grand rayon r tel que la sous-chaîne de $i - r$ à $i + r$ est un palindrome. L'algorithme naïf est le suivant. Pour tout i, on initialise $p[i] = 0$ et on incrémente $p[i]$ jusqu'à trouver le plus grand palindrome $t[i - p[i], \ldots, i + p[i]]$ centré en i. Détail : si on a rajouté des caractères sentinelles différents en début et fin de t, on peut éviter dans cette boucle le test de débordement du tableau t.

L'amélioration de Manacher concerne l'initialisation de $p[i]$. Supposons qu'on connaisse déjà un palindrome centré en c et de rayon $r - c$, donc terminant à droite en r. Soit i' le mirroir de i par rapport à c, voir Figure 2.1. Il y a une relation forte entre $p[i]$ et $p[i']$. Dans le cas où $i + p[i']$ ne dépasse pas r, on peut initialiser $p[i]$ par $p[i']$. Ceci est une opération valide, car le palindrome centré en i' de rayon $p[i']$ est inclus dans la première moitié du palindrome centré en c et de rayon $r - c$, donc il se trouve aussi dans la deuxième moitié.

Après avoir calculé $p[i]$ il faut mettre à jour c et r, pour préserver l'invariant qu'ils codent un palindrome avec r maximal. La complexité est linéaire, car chaque comparaison de caractères est responsable d'une incrémentation de r.

```
def manacher(s):
    [...]
    # tous les indices réfèrent àune chaîne fictive t
    # de la forme "^#a#b#a#a#$" si s="abaa"
    if s=="":
        t = "^#$"
    else:
        t = "^#" + "#".join(s) + "#$"
    # invariant: pour chaque préfixe vu
    # on maintient un palindrome centré en c et de bord droit r
    # qui maximise r
    # ainsi que p[i] = plus grand rayon d'un palindrome centré en i
    c = 0
    r = 0
    p = [0] * len(t)
    for i in range(1,len(t)-1):
        # -- refléter l'indice i par rapport àc
        mirror = 2*c - i
        p[i] = max(0, min(r-i, p[mirror]))
        # -- faire grossir le palindrome centré en i
        while t[i+1+p[i]] == t[i-1-p[i]]:
            p[i] += 1
        # -- ajuster centre si nécessaire
        if i+p[i] > r:
            c = i
            r = i+p[i]
    # -- extraire solution
    (k,i) = max((p[i],i) for i in range(1,len(t)-1))
    return ((i-k)/2, (i+k)/2)
```

Problèmes uva :11151 Longest Palindrome, livearchive :4975 Casting Spells

2.2 Chercher une sous-chaîne

```
entrée:  lalopalalali lala
sortie:        ^
```

Étant une chaîne s de longueur n et une chaîne t de longueur m, on veut trouver le premier indice i tel que t est une sous-chaîne de s à la position i. La réponse devrait être -1 si t n'est pas une sous-chaîne de s.

L'algorithme naïf consiste à tester tout les alignements possibles de t en dessous de s et pour chaque alignement i vérifier caractère par caractère si t correspond à $s[i..i + m - 1]$. Il a une complexité de $O(nm)$ dans le pire des cas. L'exemple suivant montre les comparaisons faits par

l'algorithme sur un exemple. Chaque ligne correspond à un choix de i, et indique les caractères qui correspondent, ou un $\times$ en cas de différence.

	0	1	2	3	4	4	5	6	7	8	9	10
	l	a	l	o	p	a	l	a	l	a	l	i
0	l	a	l	×								
1		×										
2			l	×								
3				×								
4					×							
5						×						
6							l	a	l	a		

On observe qu'après le traitement de i, on connaît déjà une bonne partie de la chaîne s. On pourrait utiliser cette information pour ne plus comparer $t[0]$ avec $s[1]$, comme dans l'exemple.

Appelons le chevauchement de deux chaînes x, y le plus long mot qui est à la fois suffixe strict de y et préfixe strict de x. Au moment de détecter une différence entre $s[i]$ et $t[j]$ on pourrait alors décaler t de telle sorte à ce que $s[0 \dots i-1]$ chevauche avec t. Comme le suffixe de $s[0 \dots i-1]$ est $t[0 \dots j-1]$ — les j derniers comparaisons ont montré égalité entre $s[i-j \dots i-1]$ et $t[0 \dots j-1]$ — le décalage qu'on va appliquer à t ne dépend que de t.

On peut donc déterminer dans un précalcul ce décalage. On note par $r[j]$ la valeur j moins le chevauchement entre $t[0 \dots j-1]$ avec lui-même. L'implémentation est montré ci-dessous. Pour analyser la complexité on distingue la partie qui calcule r avec la recherche proprement parlé. La première partie coûte $\Theta(m)$ et la deuxième $\Theta(n)$ par l'argument suivant. Chaque égalité $s[i] = t[j]$ fait augmenter j de 1 et chaque inégalité décroit strictement j, car $r[j] < j$. Or il y a au plus n égalités, et comme j ne devient jamais négatif, le nombre d'inégalités est aussi majoré par n.

```
def KnuthMorrisPratt(s, t):
   [...]
   assert t!=''
   n = len(s)
   m = len(t)
   r = [0] * m
   j = r[0] = -1
   for i in range(1,m):
      while j>=0 and t[i-1]!=t[j]:
         j=r[j]
      j+=1
      r[i]=j
   j = 0
   for i in range(n):
      while j>=0 and s[i]!=t[j]:
          j=r[j]
      j+=1
      if j==m:
         return i-m+1
   return -1
```

Variantes Avec une petite modification et sans augmenter la complexité on peut produire un tableau booléen p de taille n, qui indique pour chaque position i si t est une sous-chaîne de s à la position i. De manière plus générale on peut calculer un tableau entier p qui détermine pour chaque position i le plus grand j, tel que le préfixe de t de longueur j est une sous-chaîne de s, terminant en i.

Problèmes codility :carbo2013

Chapitre 3

Structures de données

3.1 Structures de données statiques à une dimension

Dans cette section nous considérons le problème où on reçoit un tableau de n entiers, et après un précalcul il faut rapidement répondre à des requêtes qui portent sur des intervalles. Le problème est statique dans le sens où le contenu du tableau t ne change pas.

3.2 La somme d'un intervalle

Chaque requête consiste en un intervalle d'indices $[i, j)$ et il faut retourner la somme des éléments de t entre les indices i (inclus) et j (exclu). On peut répondre à ces requêtes en temps constant après un précalcul en temps linéaire. Il suffit de calculer un tableau s de taille $n + 1$, qui contient tous les sommes des préfixes de t. Concrètement $s[j] = \sum_{i<j} t[i]$. En particulier $s[0] = 0, s[1] = t[0], s[n] = \sum t$. Alors la réponse à la requête est $s[j] - s[i]$.

3.3 Trouver un doublon dans un intervalle

Chaque requête consiste en un intervalle d'indices $[i, j]$ et il faut trouver un élément x qui apparaît au moins deux fois dans t dans cet intervalle, ou annoncer que tous les éléments sont distincts. On peut répondre à cette requête en temps constant, après un précalcul linéaire.

Par un balayage de gauche à droite, on détermine un tableau p qui indique pour chaque j le plus grand indice $i < j$ tel que $t[i] = t[j]$. Dans le cas où $t[j]$ a été vu pour la première fois on stocke $p[j] = -1$.

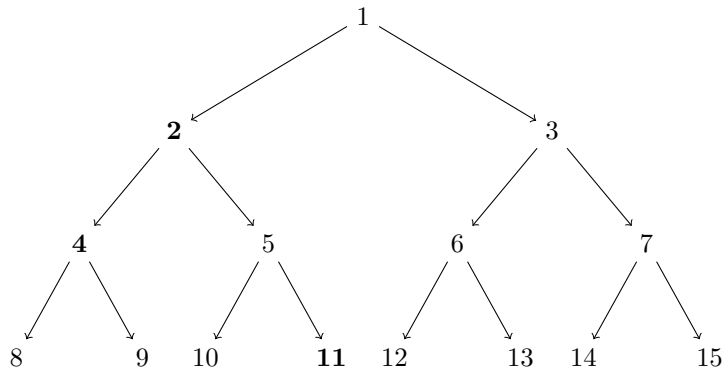
Le calcul de p nécessite de stocker pour chaque x l'indice de la dernier occurrence observée de x dans t au cours du balayage. Ceci peut être fait par un tableau si on a une promesse sur le domaine des éléments de t ou sinon par un dictionnaire basé sur table de hachage.

Puis il faut stocker parallèlement dans un tableau q, pour chaque j la plus grande valeur $p[i]$ sur tout $i \leq j$. Ainsi pour répondre à la requête $[i, j]$, si jamais $q[j] < i$ alors tous les éléments dans $[i, j]$ sont distincts, sinon $t[q[j]]$ est un elément double.

Pour déterminer les indices des deux occurrences, il suffit de calculer en parallèle avec $q[j]$ l'indice i qui a réalisé le maximum.

Problèmes baylor :5881 Unique Encryption Keys

3.4 ancêtre commun le plus bas




```
entrée: 4, 11
sortie: 2
```

Étant donnée un arbre de n noeuds on veut pouvoir répondre en temps $O(\log n)$ à des requêtes de l'ancêtre commun le plus bas (Least common ancester ou LCA en anglais). Étant données deux noeuds u, v dans l'arbre il s'agit de trouver un noeud a qui est un ancêtre commun de u, v et tel qu'aucun des fils des a n'a cette propriété.

L'idée est d'augmenter chaque noeud u avec une information de niveau *level* et des références vers des ancêtres, où *anc[k]* est un ancêtre de u de niveau *level[u]*-2^k, s'il existe et -1 sinon. Ainsi on peut utiliser ces pointeurs pour rapidement remonter aux ancêtres.

Considérons la requête *LCA(u,v)*, avec sans perte de généralité *level[u]*$\leq$ *level[v]*. D'abord on va ramener v au même niveau que u. Puis itérativement pour chaque k de $\log_2 n$ à 0, si *anc[u,k]*$\neq$*anc[v,k]* alors on remplace u, v par leur ancêtres *anc[u,k]*,*anc[v,k]*. À la fin, $u = v$ et on a trouvé l'ancêtre commun.

```
def initLCA(prec, nodes, level):
    '''met en place les données pour LCA.
    prec est un tableau qui associe àchaque noeud son père dans
    l'arbre avec la convention prec[racine]=-1 OU prec[racine]=racine.
    nodes est l'ensemble des noeuds de l'arbre.
    précalcul en temps O(n log n).
    [...]
    '''
    global anc, maxlevel
    maxlevel = log2ceil(max(level[u] for u in nodes))
    anc = [[0 for k in range(maxlevel,-1,-1)] for u in nodes]
    for u in nodes:
        anc[u][0] = prec[u]
    for k in range(1,maxlevel+1):
        for u in nodes:
            anc[u][k] = anc[ anc[u][k-1] ][k-1]

def queryLCA(u,v,level):
    '''retourne l'ancêtre commun àu,v le plus bas.
    level un tableau qui donne àchaque noeud son niveau dans l'arbre.
    complexité en temps O(log n).
    '''
    # -- supposer que v n'est pas plus haut que u dans l'arbre
    if level[u]>level[v]:
        u,v = v,u
```

```
    # -- ramener v au même niveau que u
    for k in range(maxlevel, -1,-1):
        if level[u] <= level[v]-(1<<k):
            v = anc[v][k]
    assert level[u]==level[v]
    if u==v: return u
    # -- remonter jusqu'à l'ancêtre commun le plus bas
    for k in range(maxlevel, -1,-1):
        if anc[u][k] != anc[v][k]:
            u = anc[u][k]
            v = anc[v][k]
    assert anc[u][0]==anc[v][0]
    return anc[u][0]
```