LI325 - Solutionnaire des TD456 - Programmation dynamique

Simon Arame

Printemps 2010

1 Pour débuter

1.1.1 a) $S(i)$ est la somme maximale de case consécutive qui se termine en i
1.1.1 b) $S(i) = max(S(i-1) + T(i), T(i))$
1.1.1 c) la réponse est $MAX_{i \in I} S(i)$
1.1.1 d)

```
ALG2.1.1(T)
  S <- CreerTableau(L)
  S(1) <- T(1)
  max <- 0
  POUR i de 2 à L
    S(i) <- max( S(i-1)+T(i) , T(i) )
    SI MAX < S(i)
      MAX <- S(i)
    FIN SI
  FIN POUR
  RETOURNER MAX
FIN
```

1.1.1 e) $\Theta(n)$
1.1.1 *f) enrichissement*, le même algorithme en récursif :

```
ALG2.1.1F(T)
  S <- CreerTableau(L) /* VARIABLE GLOBALE */
  S(1) <- T(1)
  n <- longueur(T)
  RETOURNER ALG2.1.1REC(n)
FIN
ALG2.1.1REC(i)
  SI S(i) == NIL
    S(i) <- max( ALG2.1.1REC(i-1)+T(i) , T(i) )
  FIN SI
  RET S(i)
FIN
```

1.1.2

```
ALG2.1.1.2(T)
  S <- CreerTableau(L)
  ind <- CreerTableau(L)
  S(1) <- T(1)
```

```
  max <- T(1)
  imax <- 1
  POUR i de 2 à L
    SI T(i) > S(i-1)+T(i)
      S(i) <- T(i)
      ind(i) <- i
    SINON
      S(i) <- T(i) + S(i-1)
      ind(i) <- ind(i-1)
    FIN SI
    SI S(i) > max
      max <- S(i)
      imax <- i
    FIN SI
  FIN POUR
  RETOURNER COUPLE(ind(imax),imax)
FIN
```

La complexité est la même : $\Theta(n)$

1.2.1 a) S(i) est la longueur de la plus longue sous-suite croissante se terminant en i (potentiellement disjointe)
1.2.1 b) $S(i) = MAX_{k<i \wedge T[k]<T[i]} S(k)$
1.2.1 c) $MAX_i S(i)$
1.2.1 d)

```
ALGO2.1.2.1(T)
  max <- 1
  S <- CreerTableau(L)
  S[1] <- 1
  POUR i de 2 à n
    smax <- 0
    imax <- nil
    POUR k de 1 à i-1
      SI smax < S[k] ET T[i] > T[k]
        smax <- S[k]
        imax <- k
      FIN SI
      SI imax != nil
        S[i] <- smax +1
      FIN SI
      SI S[i] > max
        max <- S[i]
      FIN SI
    FIN POUR
  FIN POUR
  RETOURNER max
FIN
```

1.2.1 e) La complexité est en n^2
1.2.1 *f) enrichissement*, voici le même algorithme en version recursive

```
ALG2.1.2.1F(T)
  max <- 1
  S <- CreerTableau(L,nil) /*VARIABLE GLOBALE*/
  S[1] <- 1
  ALG2.1.2.1REC(L)
  POUR i de 1 à n
    SI max < S[i]
      max <- S[i]
    FIN SI
  FIN POUR
FIN
ALG2.1.2.1REC(i)
  SI S(i) == nil
    smax <- 0
    POUR k de 1 à i-1
      SI T[k] < T[i] ET smax < S[k]
        smax <- s[k]
      FIN SI
    FIN POUR
  SINON
    RETOURNER S(i)
FIN
```

1.2.2

```
ALG2.1.2.2(T)
  max <- 1
  S <- CreerTableau(L)
  S[1] <- 1
  U <- creerTableau(L)
  U[1] <- listeVide
  imax <- 1
  POUR i de 2 à n
    smax <- 0
    indmax <- nil
    POUR k de 1 à i-1
      SI T[k] < T[i] ET smax < S[k]
        smax <- s[k]
        indmax <- k
      FIN SI
    FIN POUR
    SI indmax != nil
      S[i] <- smax + 1
    FIN SI
    SI indmax != 0
      U[i] <- U[indmax] ## (i) /*## est un opérateur de concatenation*/
    SINON
      U[i] <- creerListe(i)
    FIN SI
    SI max < S[i]
      max <- S[i]
      imax <- i
    FIN SI
  FIN POUR
```

```
  RETOURNER U[imax]
FIN
```

La complexité n'est pas modifiée, elle est toujours de n^2.

1.3 Voici un algorithme naif pour trouver la plus longue sous chaine commune à 2 chaines.

```
ALG2.1.3NAIF(X,Y)
  max <- 0
  POUR i de 1 à L(X)
    POUR j de 1 à L(Y)
      l <- 0
      TANT QUE(X[i] == Y[j])
        i++
        j++
        l++
        SI max<l
          max <- l
        FIN SI
      FIN TANT QUE
    FIN POUR
  FIN POUR
  RETOURNER max
FIN
```

Cette version simple et naive n'utilise pas la programmation dynamique et est de l'ordre de L(x)*L(y)
1.3.1 a) Une sous structure optimal est S tel que $S[i][j]$ soit la longueur de la plus grande sous chaine commune se terminant à X[i] et à Y[j].
1.3.1 b)

$$S[i][j] = \begin{cases} S[i-1][j-1]+1 & \text{si } X[i] == Y[j] \\ 0 & \text{sinon} \end{cases} \quad (1)$$

1.3.1 c) $MAX_{i,j}S[i][j]$
1.3.1 d)

```
ALG2.1.3.1(X,Y)
  n <- L(X)
  m <- L(Y)
  S <- creerTableau(n,m,0)
  max <- 0
  POUR i de 1 à n
    POUR j de 1 à m
      SI X[i] == Y[j]
        SI i==1 OU j==1
          S[i][j] = 1
        SINON
          S[i][j] <- S[i-1][j-1] + 1
      SINON
        S[i][j] <- 0
      FIN SI
      SI max < S[i][j]
        max <- S[i][j]
```

```
      FIN SI
    FIN POUR
  FIN POUR
  RETOURNER max
FIN
```

1.3.1 e) l'algorithme est de l'ordre de nm
1.3.2

```
MEME PROCEDURE QUE ALG2.1.3.1 À l'EXCEPTION DE
1-L'AJOUT D'UNE VARIABLE INDICE_MAX(A,B) MISE À JOUR AU MÊME CONDITION QUE MAX COMME SUIT :
    INDICE_MAX <- COUPLE(I,J)
2-L'OBJET RETOURNER EST DONC TRIPLET(INDICE_MAX.A,INDICE_MAX.B,MAX)
```

La complexité n'est pas modifiée

1.4.1 a) la sous structure optimale est S tel que $S[i]$ soit la valeur obtenue avec un poid de i unité
1.4.1 b)
$S[p] = MAX_{j|poids[j]\leq P}(S(pmax - poids[j]) + val[j])$
1.4.1 c) la réponse se trouve en $S[pmax]$
1.4.1 d)

```
ENTRÉES :
  p le tableaux des poids des objets
  pmax le poid maximum que l'on peut emporter
  v les valeurs des objets
  n le nombre d'objets
ALG2.1.4.1(p,v,n,pmax)
  S[0..pmax] <- init(0)
  POUR i de 1 à pmax
    max <- 0
    POUR j de 1 à n
      SI p[j] <  i & max < S[i-p[j]]+v[j]
        max <- S[i-p[j]]+v[j]
      FIN SI
    FIN POUR
    S[i] <- max
  FIN POUR
  RETOURNER S[pmax]
FIN
```

1.4.1 e) La complexité est de l'ordre de pn
1.4.2

```
ENTRÉES :
  p le tableaux des poids des objets
  pmax le poid maximum que l'on peut emporter
  v les valeurs des objets
  n le nombre d'objets
ALG2.1.4.1(p,v,n,pmax)
  S[0..pmax] <- init(0)
  obj[0..n] <- init
  POUR i de 1 à pmax
```

```
    max <- 0
    obj_p <- 0
    POUR j de 1 à n
      SI p[j] <  i & max < S[i-p[j]]+v[j]
        max <- S[i-p[j]]+v[j]
        obj_p <- j
      FIN SI
    FIN POUR
    S[i] <- max
    obj[i] <- obj_p
  FIN POUR
  liste <- listeVide()
  po <- pmax
  TANT QUE po != 0
    liste <- liste UNION {obj[po]}
    po <- po - p[obj[po]]
  FIN TANT QUE
  RETOURNER COUPLE(S[pmax],liste)
FIN
```

La complexité n'est pas modifiée.

1.5.1 a) la sous structure optimale est S tel que $S[i]$ soit le sous-ensemble maximale de l'arbre ayant pour racine i
1.5.1 b)
$S[i] = MAX(\sum_{f \in \text{fils}(i)} S[f], \sum_{pf \in \text{fils(fils}(i))} S[pf] + 1)$
1.5.1 c) la réponse se trouve en $S(\text{r}(A)$
1.5.1 d)

```
ALG2.1.5.1(A)
  n<- Longueur(A)
  S <- creerTableau(n,nil) /*VARIABLE GLOBALE*/
  S[0] <- 0
  Aux2.1.5.1(racine(A))
  RETOURNER S[racine(A)]
FIN
AUX2.1.5.1(i)
  SI (S[i] == nil)
    S1 <- 0
    S2 <- 0
    POUR TOUT LES FILS f DE i
      S1 <- S1 + Aux2.1.5.1(f)
    FIN POUR
    POUR TOUT LES PETITFILS pf DE i
      S2 <- S2 + Aux2.1.5.1(pf)
    FIN POUR
    S[i] <- MAX( S1, S2+1 )
  FIN SI
  RETOURNER S[i]
FIN
```

1.5.1 e) la complexité est de l'ordre du nombre de noeuds
1.5.2

```
ALG2.1.5.2(A)
  n<- Longueur(A)
  S <- creerTableau(n,nil) /*VARIABLE GLOBALE*/
  S[0] <- 0
  C <- creerTableau(n,"")
  Aux2.1.5.1(racine(A))
  RETOURNER ListeNoeud(racine(A))
FIN
AUX2.1.5.2(i)
  SI (S[i] == nil)
    S1 <- 0
    S2 <- 0
    POUR TOUT LES FILS f DE i
      S1 <- S1 + Aux2.1.5.1(f)
    FIN POUR
    POUR TOUT LES PETITFILS pf DE i
      S2 <- S2 + Aux2.1.5.1(pf)
    FIN POUR
    SI S1>S2
      S[i] <- S1
      C[i] <- "f"
    SINON
      S[i] <- S2+1
      C[i] <- "pf"
    FIN SI
  FIN SI
  RETOURNER S[i]
FIN
ListeNoeud(i)
  //cas de base
  SI i==0
    RETOURNER listeVide()
  SINON
    l <- listeVide()
    SI C[i] == "f"
      POUR TOUT LES FILS f de i
        l <- l UNION ListeNoeud(f)
      FINPOUR
    SINON
      POUR TOUT LES PETITFILS pf de i
        l <- l UNION ListeNoeud(pf)
      FIN POUR
      l <- l UNION i
    FIN SI
    RETOURNER l
  FIN SI
```

La complexité n'est pas modifié dans son ordre de grandeur mais de façon absolu, il y a un deuxième parcours de l'arbre à faire.

1.6.1 a) Un tableau C de booléen tel que $C[i] =$ vrai si la chaine $s[1..i]$ est un texte lisible.
1.6.1 b) $\vee_{j<i}(C[j] \wedge \text{dict}(j+1, i))$ 1.6.1 c) la réponse se trouve en $C[n]$
1.6.1 d)

```
ALG2.1.6.1(s)
  n <- longueur(s)
  C <- creerTableau(n)
  C[0] <- #true
  POUR i de 1 à n
    j <- 0
    b <- #faux
    TANT QUE j<i et NON(b)
      b <- C[j] ET dict(j+1,i)
      j++
    FIN TANT QUE
    C[i] <- b
  FIN POUR
  RETOURNER C[n]
FIN
```

1.6.1 e) La complexité est en n^2
1.6.2 Pour obtenir les positions des espaces, l'algorithme renvoie une liste de ces positions nommée coupure

```
ALG2.1.6.2(s)
  n <- longueur(s)
  C <- creerTableau(n)
  C[0] <- #true
  pos <- creerTableau(n)
  POUR i de 1 à n
    j <- 0
    b <- #faux
    TANT QUE j<i et NON(b)
      b <- C[j] ET dict(j+1,i)
      j++
    FIN TANT QUE
    pos[i] <- j-1
    C[i] <- b
  FIN POUR
  SI C[n]
    i <- n
    coupure <- listeVide()
    TANTQUE i!= 0
      coupure <- coupure UNION {pos[i]}
      i <- pos[i]
    FIN TANT QUE
    RETOURNER COUPLE(coupure,#true)
  SINON
    RETOURNER COUPLE(null,#false)
  FIN SI
FIN
```

La complexité est toujours de l'ordre de n^2

2 Ordonnancement

2.1.1
$C_j^1 = min(C_{j-1}^1 + a_{1,j-1}, C_{j-1}^2 + a_{2,j-1} + t_{2,j-1})$
$C_j^2 = min(C_{j-1}^2 + a_{2,j-1}, C_{j-1}^1 + a_{1,j-1} + t_{1,j-1})$

2.2.1.2
$f^* = min(f_{1,n} + x_1, f_{2,n} + x_2)$
Et la relation de récurence pour $f_{i,j}$:
$f_{1,j} = min(f_{1,j-1} + a_{1,j}, f_{2,j-1} + a_{1,j} + t_{2,j-1})$
$f_{2,j} = min(f_{2,j-1} + a_{2,j}, f_{1,j-1} + a_{2,j} + t_{1,j-1})$
Ou encore, de façon plus générale :
$f_{i,j} = min(f_{i,j-1} + a_{i,j}, f_{(3-i),j-1} + a_{i,j} + t_{(3-i),j-1}$
2.2.1.3 La complexité d'un algorithme récursif calculant f^* est donné par la relation de récurrence suivante : $T(n) = 2T(n-1) + O(1)$, on résoud cette relation et l'on obtient $T(n) = 2^n$. Voici l'algorithme utilisant la programmation dynamique qui sera donc meilleur :

```
ALG2.2.1.3(e1,e2,x1,x2,a1,a2,t1,t2)
  f1 <- creerTableau(1..n)
  f2 <- creerTableau(1..n)
  f1[1] <- a1[1] + e1
  f2[1] <- a2[1] + e2
  POUR i de 2 à n
    f1[i] <- min( f1[i-1] + a1[i] , f2[i-1] + a1[i] + t2[i-1] )
    f2[i] <- min( f2[i-1] + a2[i] , f1[i-1] + a2[i] + t1[i-1] )
  FIN POUR
  RETOURNER min (f1[n]+x1,f2[n]+x2)
FIN
```

Dans le pire des cas, la complexité est effectivement de l'ordre de n.
2.1.4

```
ALG2.2.1.3(e1,e2,x1,x2,a1,a2,t1,t2)
  f1 <- creerTableau(1..n)
  p1 <- creerTableau(2..n)
  f2 <- creerTableau(1..n)
  p2 <- creerTableau(2..n)
  f1[1] <- a1[1] + e1
  f2[1] <- a2[1] + e2
  POUR i de 2 à n
    SI f1[i-1] + a1[i] < f2[i-1] + a1[i] + t2[i-1]
      f1[i] <- f1[i-1] + a1[i]
      p1[i] <- 1
    SINON
      f1[i] <- f2[i-1] + a1[i] + t2[i-1]
      p1[i] <- 2
    FIN SI
    SI f2[i-1] + a2[i] < f1[i-1] + a2[i] + t1[i-1]
      f2[i] <- f2[i-1] + a2[i]
      p2[i] <- 1
    SINON
      f2[i] <- f1[i-1] + a2[i] + t1[i-1]
      p2[i] <- 2
    FIN SI
  FIN POUR
  SI (f1[n]+x1 < f2[n] + x2)
    temp <- 1
  SINON
    temp <- 2
```

```
  FIN SI
  L <- listeVide()
  POUR i de n à 2
    si temp ==1
      temp<- p1[i-1]
      L <- L UNION S1i
    SINON
      temp<- p2[i-1]
      L <- L UNION S2i
    FIN SI
  FIN POUR
  RETOURNER L
FIN
```

La complexité est toujours de l'ordre de n

3 Géométrie algorithmique

3.1.1 Soit v un sommet d'un polygone de n sommet de n côtés. Pour toute triangulation, $v1$ peut-être cordé avec $v_3, \ldots, vn-1$
On sépare avec une corde un polygone de n sommets en 2 parties de p et de q sommets, les sommets aux extrémités de la corde étant dans chacune des parties de tellle sorte que $p+q-2=n$. Soit $c(i)$ le nombre de cordes pour un polygone de i sommets, on a alors :
$c(n) = 1 + c(p) + c(q)$
$c(n) = 1 + (p-3) + (q-3)$
$c(n) = (p+q-2) - 3$
$c)n) = n-3 =>$ CQFD1
Et soit $t(i)$ le nombre de triangle d'un polygone cordé de i sommets, on a alors :
$t(n) = t(p) + t(q)$
$t(n) = (p-2) + (q-2)$
$t(n) = (p+q-2) - 2$
$t(n) = n-2 =>$ CQFD2
3.1.2 La formule reliant la pondération totale de T à la pondération $w(v_1, v_k, v_n)$ va comme suit :
$T = t[1,n] = t[1][k] + t[k][n] + w(v_1, v_k, v_n)$
Et, donc, voici la relation de récurence que l'on déduit :
$t[i,j] = min_{i+1 \leq k \leq j-1}(t[i][k] + t[k][j] + w(v_i, v_k, v_j))$
Pour les sous questions 3.1.3, 3.1.4, 3.1.5 ainsi que la section 4, merci de me faire de vos propositions de solutions à simonarame AT xsimo.com

4 Problème NP-Complet

5 Bio-informatique

5.1 Pour cette exercice, on considère que les même sous questions sont posés comme précédemment.
5.1.1 a) La sous-structure optimale est $T[][]$ tel que $T[i][j]$ soit la pondération maximale de l'alignement de $X[1..i]$ et de $Y[1..j]$.
5.1.1 b) Caractérisons la cette sous structure optimale par une équation :

$$T[i][j] = \begin{cases} Max(T[i-1][j-1]+1, T[i][j-1]-2, TY[i-1][j]-2) & \text{si } X[i] = Y[j] \\ Max(T[i-1][j-1], T[i][j-1]-2, T[i-1][j]-2) & \text{si } X[i]! = Y[j] \end{cases} \quad (2)$$

5.1.1 c) la réponse se trouvera en T[longueur(X)][longueur(Y)]
5.1.1 d)

```
ALG2.5.1.1(X,Y)
  nx <- Longueur(X)
  ny <- Longueur(Y)
  T <- CreerTableau(nx,ny)
  T[0][0] = 0
  POUR i de 1 à nx
    T[i][0] <- -2i
  FIN POUR
  POUR i de 1 à ny
    T[0][j] <- -2j
  FIN POUR
  POUR i de 1 à nx
    POUR j de 1 à ny
      SI X[i] = Y[j]
        a <- T[i-1][j-1] + 1
      SINON
        a <- T[i-1][j-1]-1
      FIN SI
      T[i][j] <- max(a,T[i][j-1]-2,T[i-1][j]-1)
    FIN POUR
  FIN POUR
  RETOURNER T[nx][ny] /*La pondération du meilleur alignement*/
FIN
```

La complexité est de l'ordre de $nx * ny$
5.1.2 Si l'on veut cette fois conserver les indices des espaces insérés :

```
ALG2.5.1.1(X,Y)
  nx <- Longueur(X)
  ny <- Longueur(Y)
  T <- CreerTableau(nx,ny)
  P <- CreerTableau4-UPLET_A,B,C,D_(nx,ny)
  T[0][0] = 0
  POUR i de 1 à nx
    T[i][0] <- -2i
    P[i][0] <- 4-UPLET(i-1,0,x[i]," ")
  FIN POUR
  POUR i de 1 à ny
    T[0][j] <- -2j
    P[0][j] <- 4-UPLET(0,j-1," ",Y[j])
  FIN POUR
  POUR i de 1 à nx
    POUR j de 1 à ny
      SI X[i] = Y[j]
        a <- T[i-1][j-1] + 1
      SINON
        a <- T[i-1][j-1]-1
      FIN SI
      T[i][j] <- max(a,T[i][j-1]-2,T[i-1][j]-1)
      SI T[i][j] = a
```

```
        P[i][j] <- 4-UPLET(i-1,j-1,X[i],Y[j])
      SINON
        SI T[i][j] = T[i-1][j]
          P[i][j] <- 4-UPLET(i-1,j,X[i]," ")
        SINON
          P[i][j] <- 4-UPLET(i,j-1," ",Y[j])
        FIN SI
      FIN SI
    FIN POUR
  FIN POUR
  L <- ListeVideCOUPLE_A,B_
  ix <- nx
  iy <- ny
  TANT QUE ix != 0 OU iy != 0
    L <- L UNION COUPLE(P[i][j].C,P[i][j].D)
    ix <- P[i][j].A
    iy <- P[i][j].B
  FIN TANT QUE
/* On retourne la liste de couple (A,B)
tel que L[i].A =" "  si X[i] = " " dans le meilleur alignement*/
 RETOURNER L

FIN
```