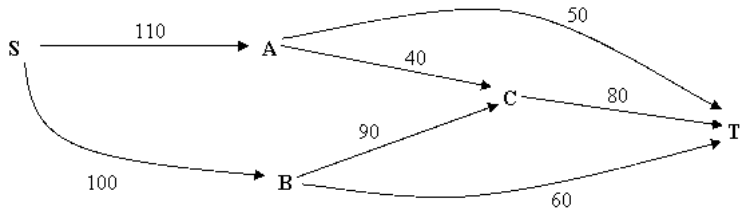
Algorithmique — M1

Examen du 11 janvier 2010

Corrigé

On applique un algorithme de cours

Exercice 1 – *Flux maximum*



Pour le réseau ci-dessus on cherche à trouver le flux (flot) maximum en appliquant un algorithme de cours.

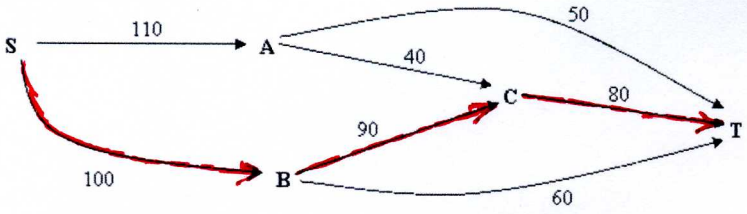
1. Choisissez un algorithme (écrivez juste son nom s'il s'agit d'un algorithme connu).

   Correction. Ford-Fulkerson.

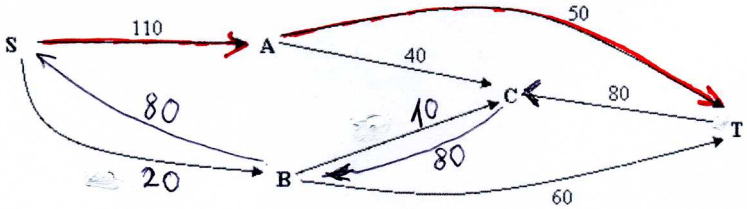
2. Appliquez l'algorithme (dessinez toutes ses itérations).
3. Donnez le résultat final : flux maximum et sa valeur.

Correction. Voir page suivante.

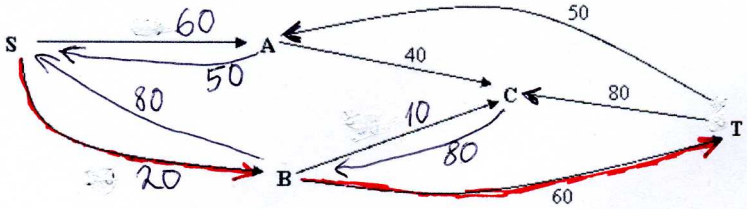
Réseau initial :



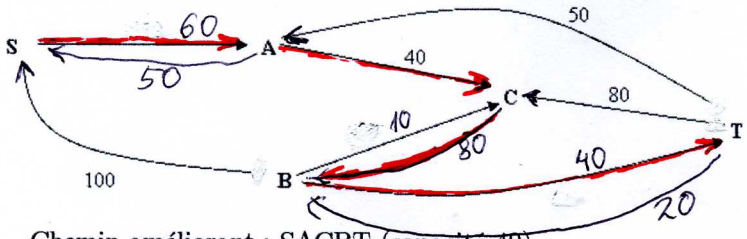
Chemin améliorant : SBCT (capacité 80).
Réseau résiduel :



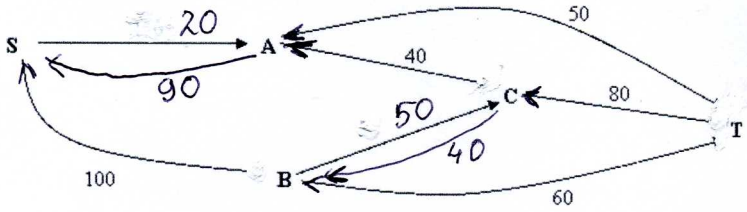
Chemin améliorant : SAT (capacité 50).
Réseau résiduel :



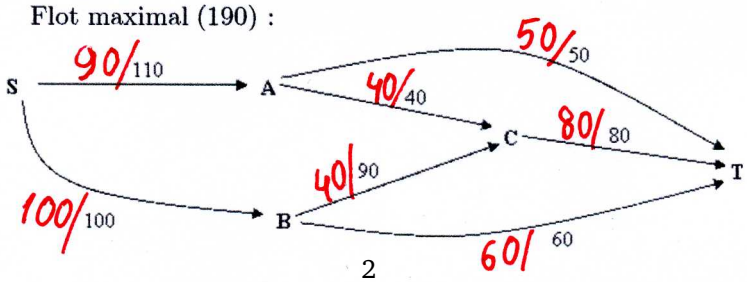
Chemin améliorant : SBT (capacité 20).
Réseau résiduel :



Chemin améliorant : SACBT (capacité 40).
Réseau résiduel :



Il n'y a plus de chemin améliorant.
Flot maximal (190) :



On adapte un algorithme de cours

Exercice 2 – *32 cavaliers*

On cherche à disposer 32 cavaliers sur l'échiquier 8x8 pour qu'ils ne soient pas en prise (wikipedia dit que c'est possible). On cherche à développer un algorithme de type backtracking qui trouve une telle disposition. Par souci d'efficacité on souhaite voir chaque disposition une seule fois et non dans tous les ordres possibles.

1. Écrivez l'algorithme (en pseudocode)

Correction. La position d'un cavalier sera représentée par un entier de 0 à 63. L'algorithme de type retour-arrière utilisera un tableau de positions de cavaliers CAV[1..32]. Initialement ce tableau est vide, ensuite k premières positions seront remplies par des entiers en ordre croissant représentant k cavaliers. La fonction suivante cherche la disposition de cavaliers en utilisant la recherche en profondeur. On appelle cette fonction quand k cavaliers sont déjà placés sur le tableau. Si $k = 32$ on s'arrête, sinon on cherche à placer encore un cavalier.

```
fonction cavaliers(k)
        si k==32
          imprimer CAV ; arrêter
        si k=0
          alors début =0
          sinon début= CAV[k]+1
        pour v de début à 63
          CAV[k+1]=v
          si test(k+1)
            cavaliers(k+1)
```

La fonction test(n) vérifie que le cavalier numéro n n'est pas en prise avec les cavaliers précédents.

```
fonction booléenne test(n)
        pour i de 1 à n-1
              si enPrise (CAV[i],CAV[n])
                 retourner false
        retourner true
```

La primitive enPrise(a,b) vérifie est-ce que deux cavaliers en positions a et b sont en prise.

```
fonction booléenne enPrise(a,b)
        xa= a mod 8 // la ligne de a
        ya = a div 8// la colonne de a
        xb= b mod 8// la ligne de b
        yb = b div 8// la colonne de a
        retourner (abs(xa-xb)=1 && abs(ya-yb)=2)| |(abs(xa-xb)=2 && abs(ya-yb)=1)
```

Toutes les fonctions sont prêtes, pour trouver la disposition de cavaliers il suffit de faire l'appel à partir de la configuration vide : cavaliers(0).

2. Expliquez cet algorithme (vous pouvez vous inspirer de l'indication)

Correction. voir point précédent

3. Estimez le nombre d'opérations nécessaire

Correction. Chaque configuration est un sous-ensemble de l'échiquier (64 cases) qui contient de 0 à 32 éléments. Il existe 2^{63} tels sous-ensembles. L'analyse de chaque sous-ensemble demande 30 opération (fonction test). On arrive à 2^{68} opérations. Cette estimation est beaucoup trop pessimiste (dans l'algo on rejette une grande partie de ces sous-ensembles en trouvant de cavaliers en prise), mais trouver une estimation plus réaliste est difficile.

On invente des algorithmes

Exercice 3 – *La meilleure période*

Le bénéfice net (positif ou négatif) de la société D&Q pendant n dernières années est représenté par le tableau $\beta = (b_1, b_2, \ldots, b_n)$. Pour une période contiguë $i..j$ on définit le bénéfice cumulé $BC = (b_i + b_{i+1} + \cdots + b_j)$. On s'intéresse à la meilleure période de l'histoire de la société où le bénéfice cumulé est maximal, et surtout à la valeur BCM de ce bénéfice cumulé maximal.

1. Pour $\beta = (-5, 1, 3, -1, 10, -2, 0)$ trouvez le BCM à la main.

Correction. $1 + 3 - 1 + 10 = 13$.

2. Proposez un algorithme itératif simple ("naïf") qui calcule BCM. Estimez sa complexité.

Correction. On calcule toutes les sommes partielles et cherche leur maximum.

```
int function BCM(B,s,f)
        courant=0
        pour i de s à f
            somme=0
            pour j de i à f
                somme=somme+B[j]
                si(courant<somme)
                  courant=somme
        retourner courant
```

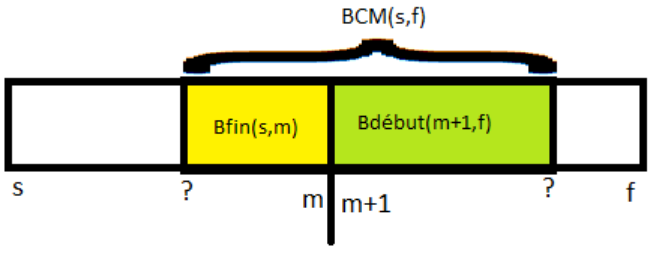
les deux boucles imbriquées donnent une complexité quadratique $O(n^2)$

3. Proposez un algorithme de type Diviser-Pour-Régner qui calcule BCM.

Indication.

- Soit la fonction $BCM(\beta, s, f)$ trouve la valeur de BCM à l'intérieur de la période $s..f$. On va programmer cette fonction.
- Coupez le tableau en deux moitiés.
- La meilleure période peut être dans la moitié gauche, dans la moitié droite ou bien à cheval entre les deux moitiés. Inventez comment traiter chacune de ces options, et comment en choisir la meilleure.
- Donnez un algorithme récursif pour $BCM(\beta, s, f)$

Correction. On suit l'indication. Pour le cas de base (s=f) on n'oublie pas que si la valeur est négative il vaut mieux prendre la période vide avec bénéfice 0. Le seul point subtil est comment trouver la meilleure période à cheval entre les deux moitiés du tableau. Pour le faire on trouvera (avec la méthode directe) la meilleure période qui se termine à m (fonction BFin) et la meilleure période qui commence à $m+1$ (fonction Bdébut) - voir le dessin.




```
fonction BCM(B,i,j)
        si i=j
            retourner max(0,B[i])
        m= (i+j) div 2
        gauche=BCM(B,i,m)
        droite=BCM(B,m+1,j)
        acheval=BFin(B,i,m)+BDebut(B,m+1,j)
        retourner max(gauche,droite,acheval)
```

Les fonctions BFin et BDebut se calculent par des boucles simples :

```
int function Bfin(B,i,m)
        courant=0
        somme=0
        pour k décroissant de m à i
            somme=somme+B[k]
            si(courant<somme)
              courant=somme
        retourner courant
```

et

```
int function Bdébut(B,M,j)
        courant=0
        somme=0
        pour k de M à j
            somme=somme+B[k]
            si(courant<somme)
              courant=somme
        retourner courant
```

4. Analysez la complexité de votre algorithme Diviser-Pour-Régner.

Correction. Le calcul pour la table de taille n se réduit à deux appels récursifs (pour la taille n/2) et au calculdes deux fonctions auxiliaires BFin et BDebut de complexité $O(n)$. Ça donne la récurrence :

$$T(n) = 2T(n/2) + O(n).$$

On applique Master Theorem : l'exposant est $\log_2 2 = 1$, la perturbation O(n) est donc moyenne. On déduit que

$$T(n) = O(n \log n).$$

Exercice 4 – *Les phrases*

Une "phrase" est une séquence quelconque de mots français collés ensemble sans espaces ni ponctuation. Par exemple

gloutonalgorithmediviserpourregnerdynamique

est une phrase, par contre

abcdefbonjour

ne l'est pas.

On suppose dans cet exercice qu'une fonction booléenne `mot(w)` est fournie. Elle renvoie `vrai` si la chaîne w est un mot français. Un appel de cette fonction prend une unité de temps.

Le problème algorithmique à résoudre dans cet exercice est le suivant : étant donné une chaîne $v = a_1 a_2 \dots a_n$ vérifier si c'est vraiment une phrase. On utilisera la programmation dynamique pour concevoir un algorithme polynômial qui résolve ce problème.

1. Soit $p(i)$ une fonction booléenne vraie si et seulement si le préfixe de v de longueur i (à savoir la sous-chaîne $a_1 a_2 \dots a_i$) est une phrase. Écrivez les équations de récurrence pour cette fonction sans oublier les cas de base.

Correction.

$$p(i) = \begin{cases} \textbf{true} & \text{si} \quad i = 0 \\ \bigvee_{k=1}^{i} (\text{mot}(a_{k..i}) \wedge p(k-1)) & \text{si} \quad i > 0 \end{cases}$$

Pour ceux qui n'aiment pas les grosses formules booléennes on peut exprimer la deuxième ligne par un texte suivant :

> La chaîne a[1..i] est une phrase (et donc $p[i] =$ **true**) si et seulement si on peut trouver une position k dans cette chaîne telle que :

–le suffixe à partir de la position k est un mot (c-à-d $mot(a[k..i] =$ **true**
–le préfixe avant la position k est une phrase (c-à-d $p(k-1) =$ **true**

2. Écrivez un algorithme efficace (récursif avec "marquage" ou itératif) pour calculer p.

Correction. En observant la récurrence pour p on constate que le tableau de valeurs p(i) peut être rempli de manière itérative dans l'ordre croissant de i. D'ou l'algorithme itératif suivant :

```
booléen P[0,n]
P[0]= true
pour i de 1 à n
      trouvé=false
      k=i // on cherche où couper
      tant que(k>0 && ! trouvé)
            trouvé = P[k-1] && mot(a[k..i])
            k= k-1
      P[i]=trouvé
```

3. En sachant calculer la fonction choisie p , comment répondre à la question initiale : est-ce que v est une phrase.

Correction. On a choisi la méthode itérative qui remplit un tableau. La réponse est dans la dernière case de ce tableau.

```
si P[n]
  Imprimer("c'est une phrase")
sinon Imprimer("ce n'est pas une phrase")
```

4. Analysez la complexité de votre algorithme.

Correction. Les deux boucles imbriquées donnent al complexité $O(n^2)$.

5. Appliquez votre algorithme à la chaîne "bonjournaliste".

Correction.
–P[0]=**true** (cas de base).
–P[1]=**false** (on ne trouve pas où couper).
–P[2]=**false** (on ne trouve pas où couper).
–P[3]=**true** (on coupe à k=1 : suffixe "bon", prefixe vide).
–P[4,5,6]=**false** (on ne trouve pas où couper).
–P[7]=**true** (on coupe à k=4 : suffixe "jour", prefixe "bon").
–P[8,9]=**false** (on ne trouve pas où couper).
–P[10]=**true** (on coupe à k=4 : suffixe "journal", prefixe "bon").
–P[11,12,13]=**false** (on ne trouve pas où couper).
–P[14]=**true** (on coupe à k=4 : suffixe "journaliste", prefixe "bon").

6. Comment modifier votre algorithme pour qu'il imprime à la fin une décomposition de la phrase en séquence de mots (si c'est possible).

Correction. On peut, par exemple stocker dans chaque case du tableau P[i] à la place du booléen **true** l'entier k désignant la position où commence le dernier mot de la décomposition de a[1..i]. Si la décomposition est impossible P[i]=0.
L'algorithme de calcul du tableau P change très peu :

```
int P[0,n]
P[0]= 9999
pouri de 1 à n
      trouvé=false
      k=i // on cherche où couper
      tant que(k>0 && ! trouvé)
            trouvé = (P[k-1]>0) && mot(a[k..i])
            k= k-1
      P[i]=k+1
```

Après avoir calculé les valeurs de P[i] il est facile d'imprimer la phrase jusqu'à la position i avec la fonction récursive :

```
fonction PrettyPrint(i)
        siP[i]=0
          Imprimer("ce n'est pas une phrase")
        sinon si i=0
          //rien à imprimer
        sinon
          k= P[i]
          PrettyPrint(k-1)
          Imprimer("  ")
          Imprimer(a[k..i])
```

Dans le programme principal il faudra faire appeler PrettyPrint(n) pour imprimer la décomposition demandée.