CSI3105 Automne 2009
Devoir 2 Solution

1. Taille de l'entrée: n, le nombre de clés dans S.

   L'opération de base comptée: comparaison de x clefs dans S.

   Moyenne des cas d'analyse:

   Soit les 9 classes d'entrée I_i pour i = 1, 2, ..., 9 correspondant à celles décrites dans la question. Étant donné que toutes les classes d'entrée ont les mêmes chances, nous avons : $p(I_i) = 1/9$ pour i = 1, 2, ..., 9. En retraçant l'algorithme nous pouvons aussi obtenir les valeurs de $t(I_i)$, comme le montre le graphique ci-dessous:

i	1	2	3	4	5	6	7	8	9
$t(I_i)$	3	1	3	5	4	4	6	4	6

Ainsi, nous avons l'analyse du cas moyen comme suit:

$A(n) = \sum(p(I_i)t(I_i): i = 1, 2, ..., 9)$
$= 1/9 (3 + 1 + 3 + 5 + 4 + 4 + 6 + 4 + 6)$
$= 1/9*36$
$= 4.$

2. Nous allons utiliser les relations de log: $\lg n = (\lg e)^* (\ln n)$ .....(*)
   et $\log ab = \log a + \log b$ (**)
   et La règle de l'Hôpital .....(LH)

a) Soit $g(n) = n\lg n$, let $f(n) = kn\lg(kn)$, $k>0$. (NOTE: $k>=0$ dans le devoir c'était l'hypothèse comme mentionne aussi en classe)

$$\lim_{n\to\infty} \frac{g(n)}{f(n)}$$

$$= \lim_{n\to\infty} \frac{n\lg n}{kn\lg kn}$$

$$= \lim_{n\to\infty} \frac{\lg n}{k\lg kn} \quad \text{.....simplification}$$

$$= \lim_{n\to\infty} \frac{(\lg e)(\ln n)}{(k)(\lg k + \lg n)} \quad \text{.....utilisation de (**)}$$

$= \lim_{n\to\infty} \frac{(\lg e)(\ln n)}{(k\lg k) + k\lg e(\ln n)}$ .....utilisation de (*)

$= \lim_{n\to\infty} \frac{(\lg e)/n}{(k\lg e)/n}$ .....utilisation de (LH)

$= \lim_{n\to\infty} \frac{1}{k}$. .....simplification

$= 1/k$. comme $1/k$ est constant et >0, nous avons $g(n) = \Theta(f(n))$ comme demandé.

b) Soit $g(n) = 6n^2 + 20n$, let $f(n) = n + 5$.

$\lim_{n\to\infty} \frac{g(n)}{f(n)}$

$= \lim_{n\to\infty} \frac{6n^2 + 20n}{n + 5}$

$= \lim_{n\to\infty} \frac{12n + 20}{1}$ .....utilisation (LH)

$= \infty$. Alors $g(n) = \Omega(f(n))$ comme d demandé.

3.a)

Analyse du pire des cas (en supposant n est une puissance de 2, soit $n = 2^k$):

Taille de l'entrée: n, le nombre d'entiers dans X

Les opérations de base compté: Additions impliquant X et en utilisant la fonction max Entrée de taille n qui donne le pire des cas: Tous les cas sont les mêmes

Analyse:

Si n> 1: Dans ce cas, nous avons deux pour les boucles, chacune avec deux opérations de base par itération, et chaque boucle « for » exécute n/2 fois, pour un total de 2n opérations de base. Nous avons également deux autres opérations de base à compter (à la ligne Maxcrossing ..., et la dernière déclaration en ligne return...) qui se produisent en dehors des boucles « for », ainsi que les opérations de base qui surviennent dans les deux appels récursifs à l'algorithme MaxSum. Chaque appel récursif à MaxSum consiste à examiner une série de n/2 nombre dans X, et chacun a besoin W (n/2) opérations de base. Ainsi, au total, nous avons:

$$W(n) = 2n + 2 + 2W(n/2).$$

Si n = 1: Dans ce cas (qui est le cas de base), nous avons juste une opération max et donc

$W(1) = 1$.

Ainsi, la relation de récurrence pour W (n) est:

$W(n) = 2W(n/2) + 2n + 2$ for $n > 1$

$W(1) = 1$.

(Remarque: L'algorithme n'atteint jamais le cas n = 0 récursivement, et 0 n'est pas une puissance de 2, donc nous ne considérons pas n = 0)

Résoudre la relation de récurrence utilisant la substitution arrière (rappeler $n = 2^k$):

$W(2^k) = 2W(2^{k-1}) + 2*2^k + 2$

$= 2(2W(2^{k-2}) + 2*2^{k-1} + 2) + 2*2^k + 2$

(comme $W(2^{k-1}) = 2W(2^{k-2}) + 2*2^{k-1} + 2$ Par la relation de récurrence)

$= 2^2*W(2^{k-2}) + 2*2^{k+1} + 2^2 + 2$ (simplifiant et regroupant)

.

. (A noter qu'on ne pouvait faire une substitution de plus ici si vous n'êtes pas sure du modelé encore)

.

$= 2^i *W(2^{k-i}) + i*2^{k+1} + 2^i + 2^{i-1} + ... + 2$

.

.

.

$= 2^k *W(2^{k-k}) + k*2^{k+1} + 2^k + 2^{k-1} + ... + 2$ (quand i=k)

$= 2^k + k*2^{k+1} + \Sigma(2^j$: for j = 1 to k) (utilisant W(1) = 1)

$= 2^k + k*2^{k+1} +((2^{k+1} - 1) – 2^0)$ (Utilisant la formule de sommation, , Appendix A. A noter que nous devons soustraire 2^0 ici, comme la formule commence avec j=0)

$= n + (\lg n)*2n + 2n -1 -1$ (utilisant $n=2^k$, $k = \lg n$)

$= 2n\lg n + 3n -2$ (simplification)

On peut (et doit) confirmer notre résultat pour la solution de la relation de récurrence à l'aide de l'induction.

Revendication: $W(n) = 2n\lg n + 3n -2$ pour $n >= 1$, n est une puissance de 2.
Preuve: (par induction)
Cas de base: Si n = 1, puis par la relation de récurrence, W(1) = 1, et par notre formule, $W(1) = 2\lg 1 + 3 - 2 = 1$. Donc l'assertion est vraie pour n = 1.

Hypothèse d'induction: Supposons que l'assertion est vraie pour tout 1 <= m <n, m est une puissance de 2, à savoir

$W(m) = 2m\lg m + 3m - 2.$

Maintenant, considérons tout n> 1, n une puissance de 2. Nous avons :

$W(n) = 2W(n/2) + 2n + 2$ (par la relation de récurrence)
$= 2(2(n/2)\lg(n/2) + 3(n/2) - 2) + 2n + 2$ (ici nous devons utiliser le fait que n/2 est une puissance de 2 et <n et >=1, donc on peut utiliser l'hypothèse d'induction et substituer pour W(n/2))
$= 2n\lg(n/2) + 3n - 4 + 2n + 2$ (simplification)
$= 2n(\lg n - \lg 2) + 5n - 2$ (simplification utilisant $\lg(a/b) = \lg a - \lg b$)
$= 2n\lg n + 3n - 2$ (simplification)
Comme demandé.

b) Comme $W(n) = 2n\lg n + 3n - 2$, cet algorithme est $\Theta(n\lg n)$ quand n est une puissance de 2.

4 a)
Algorithme habituel (amélioré):
Nombre de multiplications = $n^3 = 16^3$
Nombre d'additions = $n^3 - n^2 = 16^3 - 16^2$

Nombre total d'opérations = $7(7^4) - 6(16^2) = 15{,}271$

L'algorithme de Strassen:
Nombre de multiplications = $7^{\lg n} = 7^{\lg 16} = 7^4$
Nombre d'additions / soustractions = $6*7^{\lg n} - 6n^2 = 6(7^{\lg 16}) - 6(16^2)$

Ainsi, lorsque l'on compte les opérations énumérées ci-dessus, l'algorithme habituel serait plus rapide pour n = 16.

b) Des notes de cours, la relation de récurrence pour l'algorithme de Strassen comptant les multiplications avec un seuil de 4 seront:

$W(n) = 7W(n/2)$ pour n>4, n est une puissance de 2
$W(4) = 4^3 = 64$
$W(2) = 2^3 = 8$

W(1) = 1

(Note: pour n<= 4 nous allons utiliser l'algorithme habituel pour multiplier les matrices qui demande n^3 multiplications).

5.

Entrée taille: n, le nombre de clefs dans la liste. Nous supposons $n = 2^k$ pour k> = 0.

Opérations comptées: Comparaison des clefs.

Soit K(n) représente le nombre de comparaisons pour la liste triés en ordre inverse.
(Note: Pour cette analyse, je pars du principe que les nombres sont distincts.)

Puisque la liste est triée dans l'ordre inverse, comme à chaque étape de fusion, on va fusionner deux listes où une seule liste a des valeurs qui sont tous plus petits que les autres. Compte tenu de 2 listes triées de longueur n/2 où une seule liste a des valeurs qui sont tous plus petits que ceux de la seconde liste, l'algorithme de fusion ferai n/2 comparaisons, plutôt que de n-1 comparaisons comme dans le cas le plus défavorable. Ainsi, de l'algorithme mergesort, nous avons :

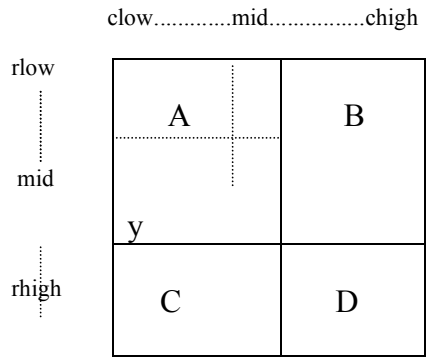
K(n) = 0 pour n = 1 (à partir du cas de base).
Pour n>1,
K(n) =(le travail du tri de la moitie gauche de la liste) + (le travail du tri de la moitie droite de la liste) +
(le travail pour combiner les deux moities de la liste)
=W(n/2) + W(n/2) + n/2
= 2W(n/2) + n/2.

6. a) Il existe plusieurs solutions pour ce problème. Ci-dessous je donne deux « diviser pour regner » solutions possibles. Les premières idées très proches de ceux utilisés en recherche binaire. Le second est plus efficace, l'algorithme Slicker qui s'exécute en temps linéaire.

Algorithme 1: Un type de recherche binaire de l'algorithme

Idée de l'algorithme:

Nous avons une large rangée rlow .... rhigh, et une colonne de range de Clow .... chigh. Pour commencer avec cette gamme sera de 1 ... n pour les lignes, et 1 ... m pour les colonnes. Nous regardons le numéro dans le tableau qui se trouve dans le milieu des deux rangées, appelez ça y. (Voir figure ci-dessous - noter que chacun des quadrants A, B, C et D auraient une taille de n/2 par n/2 pour un tableau de n par n, dans le cas où n est une puissance de 2).



Si x = y, on a fini. Si x> y, alors nous savons que X ne se trouvent pas dans le cadran A de la table (puisque les entrées dans les rangées et les colonnes sont triées). Donc, nous cherchons chacun des quadrants B, D et C de façon récursive pour X. Si x <y, alors nous savons que X ne peut pas résider dans quadrant D, et nous recherchons quadrants A, B et C de façon récursive. Enfin, si la plage de ligne ou de colonne de la rangé devient vide (c'est-à-dire nous avons rlow> rhigh ou Clow> chigh), alors nous savons x n'est pas dans le tableau.

Algorithme (en pseudo-code):

Entrée: un tableau de tableau n par m, triées comme décrit dans la question, et un nombre x.

Sortie: a 2 emplacements d'éléments du tableau, où de localisation[1] est l'emplacement des rangées de X dans le tableau, et la localisation[2] est l'emplacement de colonne de x dans le tableau. Si x n'est pas dans le tableau,Ce sont des 0.

```
void findx1(index rlow, index rhigh, index clow, index chigh, index location[])

{
  index rmid, cmid;

  if ((rlow>rhigh) || (clow>chigh)) {
        location[1]=0;
        location[2]=0;}
  else {
    rmid = ⌊(rlow + rhigh)/2⌋
    cmid = ⌊(clow + chigh)/2⌋
    if (x==table[rmid][cmid]){
          location[1] = rmid;
          location[2] = cmid; }
```

```
    else {
       if (x > table[rmid][cmid]){
          findx1(rmid+1, rhigh, clow, cmid, location);    \\ regarder dans le cadran  C
           if (location[1] = = 0) \\not found in C
              findx1(rmid+1, rhigh, cmid+1, chigh, location);  \\ regarder dans le cadran  D
              if (location[1] = = 0)  \\ not found in C or D
                  findx1(rlow, rmid, cmid+1, chigh, location); }    \\ regarder dans le
cadran  B

       else }
          findx1(rmid+1, rhigh, clow, cmid, location);    \\ regarder dans le cadran  C
           if (location[1] = = 0) \\ not found in C
              findx1(rlow, rmid, clow, cmid, location);  \\ regarder dans le cadran  A
              if (location[1] = = 0) \\ not found in C or A
                  findx1(rlow, rmid, cmid+1, chigh, location); }    \\ regarder dans le
cadran  B

    }
}
```

Analyse du cas pire (nous supposons tableau n par n, où n est une puissance de 2, soit $n = 2^k$, $k >= 0$):

Taille de l'entrée: n, le nombre de lignes et de colonnes de la table dans lesquels on cherche.
L'opération de base comptée: comparaison de x avec les clefs.

D'entrée qui donne le pire des cas: x n'est pas dans le tableau, et plus grand que tous les nombres dans le tableau

Analyse:
Chacun des quadrants a une taille de n/2 par n/2, on recherche 3 quadrants récursivement et faire 2 comparaisons en dehors des appels récursifs, nous avons donc

$W(n) = 3W(n/2) + 2$, $n>=2$
$W(1) = 2$ (Comme quand n est 1 et x est plus grand que le seul élément de la table, puis il ya 2 comparaisons, suivi de 3 appels récursives où rlow > rhigh, dont chacun nécessite 0 comparaisons avec x).

Substitution arrière:

$W(2^k) = 3W(2^{k-1}) + 2$
$= 3(3W(2^{k-2}) + 2) + 2$ (substitution)
$=3^2W(2^{k-2}) +3*2 + 2$ (simplification)

⋮

(i^{eme} ligne)
$=3^iW(2^{k-i}) + 2(3^{i-1} + 3^{i-2} + ... + 1)$

⋮

(ligne k)
$=3^kW(2^{k-k}) + 2(3^{k-1} + 3^{k-2} + ... + 1)$
$=(3^k)*2 + 2(3^{k-1} + 3^{k-2} + ... + 1)$ (utilisant $W(1) = 2$)
$=2(\Sigma (3^j: j=0 \text{ to } k))$ (simplification)
$= 2 ((3^{k+1} -1)/(3-1))$ (De l'Appendix A)
$= 3^{k+1} -1$ (simplification)
$=3* 3^k - 1$ (simplification)
$= 3*3^{lgn} - 1$ (simplification)
$= 3*n^{lg3} - 1$ (utilisant $a^{lgb} = b^{lga}$)
$\approx 3*n^{1.585} - 1$

Donc cet algorithme est $\approx\Theta(n^{1.585})$.

Algorithme 2: Un algorithme en temps linéaire.

Idée de l'algorithme: Supposons que nous ayons un tableau qui est n par m.

Nous voyons à l'entrée Y = table[n][1]. Si x = y, on a fini. Si x <y, alors nous pouvons conclure que x n'est pas à la ligne n. Donc, nous cherchons maintenant un tableau qui contient des lignes 1 à n-1, et les colonnes 1 à m (c'est-à-dire la table avec la n-ième rangée enlevée de sorte que nous avons éliminé une rangée). Si x> y, alors nous pouvons conclure que X n'est pas dans la première colonne. Donc, nous cherchons maintenant un tableau qui contient des lignes 1 à n, et les colonnes 2 à m (c'est-à-dire la table avec la 1ère colonne supprimée, de sorte que nous avons éliminé une colonne). Donc, en général, étant donné l'espace de recherche en ligne dans la gamme de 1 à lastrow, et la gamme FirstCol colonne à m, on regarde en bas à gauche l'entrée dans cet espace de recherche, à-dire Table[lastrow][FirstCol] et décider de supprimer soit une ligne (rowlast) ou une colonne (FirstCol) de l'espace. Une fois que soit le nombre de lignes ou le nombre de colonnes de la surface de la table, nous avons 0, nous nous arrêtons et de conclure que x n'est pas dans le tableau.

Pseudo-code pour l'algorithme

Entrée: un tableau de tableau n par m, triées comme décrit dans la question, et un certain nombre x.
Sortie: 2 éléments du tableau, où localisation [1] est l'emplacement des rangées de X dans le tableau, et la localisation [2] est l'emplacement de colonne de x dans le tableau. Si x n'est pas dans le tableau,ce sont des 0.

```
void findx2(index lastrow, index firstcol, index location[])
{
if (lastrow==0) || (firstcol==m+1) {
    location[1]=0;
    location[2]=0; }
else
   if (x==table[lastrow][firstcol]){
       location[1]=lastrow;
       location[2]=lastcol;}
  else {
       if (x < table[lastrow][firstcol])
            lastrow=lastrow – 1;
        else
            firstcol=firstcol+1;
        findx2(lastrow, firstcol, location)  }
}
```

Analyse du cas pire (en fait, pour le pire des cas, je n'ai pas besoin de supposer que n est une puissance de 2, alors ici je montre l'analyse en général par le tableau de n par m).

Taille de l'entrée: n+n, où n et m sont au nombre de lignes et de colonnes dans la table.

L'opération de base comptées: comparaison de x avec des entrées du tableau.

D'entrée qui donne le pire des cas: A chaque étape de la récursivité, nous éliminons une ou l'autre d'une ligne ou d'une colonne à partir de notre espace de recherche, jusqu'à ce que soit le nombre de lignes de la gauche ou le nombre de colonnes de gauche devient 0. Ainsi, on obtient la plupart des phases de la récursivité si, au tout dernier stade, autant le nombre de lignes que de colonnes est 1 (donc quand nous atteignons le cas de base, le nombre de lignes et de colonnes qui restent sont 1 et 0 ou 0 et 1). Un exemple d'une entrée qui crée cette situation est celle dans laquelle toutes les entrées dans le tableau sont 1, sauf pour la ligne n, où nous avons 2 de la position 1 à m-1, et la colonne m est remplie des 4, et la valeur de x est de 3.

Analyse:
Pour nos pires cas d'entrée, nous avons

$W(n+m) = W(n+m-1) + 2$ pour $n + m >= 2$,

car, à chaque étape de la récursivité on diminue le nombre total de lignes et de colonnes par 1, et nous faisons 2 comparaisons à l'extérieur de l'appel récursif.

Pour le cas de base, $W(1) = 0$ (c'est à dire quand nous avons le nombre de colonnes et de lignes est de 1 est 0).

Substitution arrière:

$W(n+m) = W(m+n-1) + 2$
$= W(m+n-2) + 2*2$ (Substitution)
$= W(m+n-3) + 2*3$ (Substitution)

(ligne i)
$= W(m+n-i) + 2*i$

(ligne m+n-1)
$=W(1) + 2*(m+n-1)$
$=2*(m+n-1)$. (Comme $W(1) = 0$)

Donc cet algorithme est $\Theta(n+m)$.