Solutions : Chapitre 6

Question # 1

Soient $A = [a_1, \ldots, a_n]$ et $B = [b_1, \ldots, b_m]$ deux ensembles de nombres. En considérant le problème de trouver l'intersection des deux ensembles (c'est-à-dire l'ensemble C contenant exactement les éléments commun à A et B) :

A) Élaborez un algorithme de force brute pour résoudre ce problème et déterminez son efficacité.

Solution :

Une solution est donnée par l'algorithme 1. Prenons la comparaison de deux éléments comme opération de base. Puisque les ensembles de départ ne sont pas modifiés, l'efficacité de l'algorithme dépend uniquement de la taille de l'instance qui est donnée par le nombre d'éléments total, soit $n + m$. Nous avons alors que l'efficacité de l'algorithme est donnée par l'équation ci-dessous.

$$C(n) = \sum_{i=1}^{n} \sum_{j=1}^{m} 1 = \sum_{i=1}^{n} m = nm \in \Theta(nm)$$

Si toutefois nous retirons de l'ensemble B l'élément b_j lorsque nous avons $a_i = b_j$ et si nous ajoutons un «break» pour éviter les comparaisons inutiles, nous devons alors faire l'analyse par cas. En meilleur cas, nous avons $A \subseteq B$ et nous parcourons les deux ensembles dans le même ordre. Dans ce cas, nous effectuons exactement une comparaison par élément de A et $C_{BEST}(n) \in \Theta(n)$. En pire cas, nous avons $A \cap B = \emptyset$ et nous devons parcourir les deux ensembles au complet. Dans ce cas, nous avons $C_{WORST}(n) \in \Theta(nm)$.

Algorithme 1 : IntersectionForceBrute($A = \{a_1, \ldots, a_n\}$, $B = \{b_1, \ldots, b_m\}$)

```
1 // Trouver l'intersection de deux ensembles.
2 // Entrées : Deux ensembles A et B.
3 // Sortie : Un ensemble Intersection contenant les éléments communs de A et B.
4 Intersection ← ∅
5 pour a_i ∈ A faire
6     pour b_j ∈ B faire
7         si a_i = b_j alors
8             Intersection ← Intersection ∪ {a_i}
9 retourner Intersection
```

B) *Élaborer un algorithme qui utilise le pré-triage pour résoudre ce problème et déterminez son efficacité.*

Solution[1] ***:***

Voir l'algorithme 2. Soit $C_1(n)$ *l'efficacité de l'algorithme de triage et soit* $C_2(n)$ *l'efficacité de RechercheBinaire. Pour* $f, g \in \mathbb{N}$*, nous avons*

$$C(n,m) = C_1(n) + \sum_{j=1}^{m} C_2(n) = C_1(n) + mC_2(n) \leq fn\log(n) + gm\log(n)$$

L'efficacité de l'agorithme 2 est donc en $O(\max(m,n)\log(n))$ *par la règle du maximum. Remarquez qu'il est préférable de trier le plus petit des deux ensembles pour que* $\log(n)$ *soit minimal.*

Algorithme 2 : IntersectionRechercheBinaire($A[1..n]$, $B[1..m]$)

```
1  // Trouver l'intersection de deux ensembles.
2  // Entrées : Deux tableaux de nombres A et B.
3  // Sortie : Un ensemble Intersection contenant les éléments communs de A et B.
4  Trier(A)
5  Intersection ← ∅
6  pour b_j ∈ B faire
7      k ← RechercheBinaire(A, b_j)
8      si k ≠ −1 alors
9          Intersection ← Intersection ∪ {b_j}
10 retourner Intersection
```

Voici une deuxième solution qui utilise le pré-triage.

Algorithme 3 : IntersectionPreTriage($A[1..n]$, $B[1..m]$)

```
1  Intersection ← ∅
2  Trier(A);
3  Trier(B);
4  i ← 1;
5  j ← 1;
6  tant que i ≤ n ∧ j ≤ m faire
7      tant que j ≤ m ∧ B[j] < A[i] faire
8          j ← j + 1
9      si j ≤ m ∧ B[j] = A[i] alors
10         Intersection ← Intersection ∪ {a_i}
11     i ← i + 1;
12 retourner Intersection;
```

Dans l'algorithme IntersectionPreTriage, les tris prennent un temps de $\Theta(n\log n)$ *et* $\Theta(m\log m)$*. Nous prenons l'incrémentation* $+1$ *comme opération de base. Cette opération est appelée au plus*

1. Il existe d'autres solutions pour ce problème. Entre autre, on peut obtenir une efficacité de $O(s\log s)$ où $s = max(n,m)$

n fois pour l'incrémentation de i et au plus m fois pour l'incrémentation de j. D'après la règle du maximum, le temps d'exécution de l'algorithme en pire cas est :

$$\begin{aligned} C_{WORST}(n,m) &\in \Theta(\max(n\log n, m\log m, n, m)) \\ &= \Theta(\max(n\log n, m\log m)) \end{aligned}$$

Question # 2

Soit un tableau de n nombres réels et soit s un entier. Soit le problème de déterminer si le tableau contient 2 éléments x et y tels que $s = x + y$. Par exemple, pour le tableau $[5, 9, 1, 3]$ et $s = 6$ la réponse et oui, mais pour le même tableau et $s = 7$ la réponse est non. Élaborez un algorithme pour résoudre ce problème qui a une efficacité meilleure que quadratique. C'est-à-dire que votre algorithme ne *doit* pas *s'exécuter en $\Omega(n^2)$,*

Première solution :

Algorithme 4 : Somme1($A[1..n]$, s)

```
1  // Déterminer si un tableau A contient deux éléments dont la somme donne s.
2  // Entrées : Un tableau de nombres réels A et un entier s.
3  // Sortie : Vrai ou Faux.
4  si s ≠ 0 alors
5  |   pour i = 1..n faire
6  |   |   A[i] ← A[i] − s/2
7  A ← TrierParValeurAbsolue(A)
8  pour i = 1..n − 1 faire
9  |   si A[i] = −A[i + 1] alors
10 |   |   retourner Vrai
11 retourner Faux
```

Dans le cas où $s = 0$, on a $A[i] + A[j] = s$ ssi $A[i] = -A[j]$. On trie A en ordre croissant des valeurs absolues de ses éléments ([-6,3,-3,1,3] devient donc [1,3,-3,3,-6]). Le cas pour une valeur arbitraire de s peut se ramener au cas $s = 0$ de la façon suivante : $A[i] + A[j] = s$ ssi $(A[i] - s/2) + (A[i] - s/2) = 0$.

L'efficacité de l'algorithme 4 est $O(n \log n)$ (n pour soustraire $s/2$ à tous les éléments si $s \neq 0$; $n \log n$ pour trier le tableau ; n pour vérifier l'existence d'un indice i tel que $A'[i] = -A'[i + 1]$ où A' est A trié et où $s/2$ a élé soustrait à tous ses éléments).

Deuxième solution :

Algorithme 5 : Somme2($A[1..n]$, s)

```
1  TriFusion(A)
2  pour i = 1..n − 1 faire
3  |   si RechercheBinaire(A, s − A[i]) alors retourner Vrai ;
4  retourner Faux
```

On tri le tableau A. Pour chaque élément $A[i]$, on utilise la recherche binaire pour vérifier la présence de $s - A[i]$ dans le tableau A. Un tri en $\Theta(n \log n)$ et n recherches binaires (en pire cas) donne $\Theta(n \log n + n \log n) = \Theta(n \log n)$.

Troisième solution :

Algorithme 6 : TroisièmeSolution($A[0..n-1]$, s)

```
1 TriFusion(A)
2 i ← 0
3 j ← n − 1
4 tant que i ≤ j faire
5 |   tant que j ≥ 0 ∧ A[i] + A[j] > s faire j ← j − 1 ;
6 |   si j ≥ 0 ∧ A[i] + A[j] = s alors retourner vrai ;
7 |   i ← i + 1
8 retourner faux
```

Après avoir trié le tableau A, on le parcourt avec deux itérateurs : l'itérateur i qui va de 0 à $n-1$ et l'itérateur j qui va de $n-1$ à 0. Voir l'algorithme 6. Chaque fois qu'on avance l'itérateur i d'un pas, on réduit j afin d'obtenir l'égalité $A[i]+A[j]=s$. Si $A[i]+A[j]<s$, c'est qu'il n'y a pas de valeur j permettant l'égalité. On passe donc au i suivant. L'opération de base est le changement de position des itérateurs i et j qui se produit au plus n fois. Le temps de calcul est donc dominé par le tri fusion en $\Theta(n \log n)$.

Question # 3

Élaborez un algorithme pour vérifier si un tableau $H[1 \ldots n]$ est un monceau (heap) et déterminez son efficacité.

Solution :

Voir Algorithme 7 pour la solution. Prenons la ligne 5 comme opération de base. En meilleur cas, l'algorithme s'arrête après le premier test. L'efficacité du meilleur cas est donc en $\Theta(1)$. En pire cas, le tableau H est un monceau et nous devons faire toutes les vérifications, c'est-à-dire $n-1$ comparaisons. Nous avons donc une efficacité en $\Theta(n)$ en pire cas.

Algorithme 7 : IsHeap($H[1..n]$)

```
1 // Déterminer si un tableau H est un monceau.
2 // Entrées : Un tableau de nombres H.
3 // Sortie : Un booléen.
4 pour i = 2..n faire
5 |   si H[i] > H[⌊i/2⌋] alors
6 |   |   retourner Faux
7 retourner Vrai
```

Question # 4

Trouvez le nombre minimal et le nombre maximal d'éléments qu'un monceau de hauteur h peut avoir.

Solution :

Un monceau de hauteur h ayant un nombre minimal de noeuds a le nombre maximal de noeuds pour les niveaux 0 à $h-1$ et 1 noeud sur le dernier niveau. Le nombre total de noeuds dans un tel monceau est donc :

$$n_{min}(h) = 1 + \sum_{i=0}^{h-1} 2^i = 1 + (2^h - 1) = 2^h$$

Un monceau de hauteur h ayant un nombre maximal de noeuds a le nombre maximal de noeuds pour les niveaux 0 à h. Le nombre total de noeuds dans un tel monceau est donc :

$$n_{max}(h) = \sum_{i=0}^{h} 2^i = 2^{h+1} - 1$$

Question # 5

A) *Élaborez un algorithme pour trouver et éliminer le plus petit élément d'un monceau et déterminez son efficacité.*

Solution :

La structure d'un monceau implique que le plus petit élément se retrouve toujours dans la dernière moitié du tableau (dans $H[\lfloor n/2 \rfloor + 1], \ldots, H[n]$). On peut donc trouver le plus petit élément en parcourant la seconde moitié de H en ordre inverse (ce qui prend un temps de $O(n)$). Éliminer ce plus petit élément peut se faire en échangeant cet élément avec le dernier élément $H[n]$, en diminuant la taille du tableau de 1 et finalement en percolant[2] l'ancienne valeur de $H[n]$ à partir de sa nouvelle position (ce qui prend un temps de $O(\log n)$). L'algorithme est donc $O(n)$.

B) *Élaborez un algorithme pour trouver et éliminer un élément donné d'un monceau et déterminez son efficacité.*

Solution :

Trouver un élément quelconque dans un monceau se fait en parcourant séquentiellement la totalité du tableau (en $O(n)$). On peut éliminer l'élément trouvé par la même procédure que celle décrite en (A) à l'exception qu'il faut percoler ou de tamiser[3] l'ancienne valeur de $H[n]$ à partir de sa nouvelle position selon qu'elle est supérieure à son parent ou inférieure au maximum de ses enfants (toujours en $O(\log(n))$). L'algorithme est donc en $O(n)$.

2. Percoler un élément e d'un monceau consiste à échanger e avec son parent si ce dernier a une valeur inférieure à e, et ce jusqu'à ce que e ait une valeur inférieure à son parent.

3. Tamiser un élément e d'un monceau consiste à échanger e avec le maximum de ses enfants si ce dernier a une valeur supérieure à e, et ce jusqu'à ce que e ait une valeur supérieure à ses enfants.