DS3 : Programmation en C, Gestion de la mémoire

19 janvier 2024

Vous pouvez réutiliser des résultats des questions précédentes pour une démonstration ou un code, même si la question n'a pas été traitée.

Vous pouvez introduire des fonctions, variables, types ou notations supplémentaires pour produire vos réponses.

Si, au cours de l'épreuve, vous repérez ce qui vous semble être une erreur d'énoncé, signalez-le sur votre copie et poursuivez votre composition en expliquant les raisons des initiatives que vous seriez amené·e à prendre.

Pour la pagination, vous numéroterez toutes les pages de votre composition en faisant figurer le numéro de la page actuelle à partir de 1, ainsi que le total de pages de votre composition. Vous ferez la pagination durant le temps de l'épreuve, et non après la fin de l'épreuve.

Votre nom devra figurer sur chacune de vos copies qui seront rendue imbriquées les unes dans les autres.

Les questions difficiles sont indiquées par une ou plusieurs étoiles (⋆). Cette difficultés est principalement indicative.

Prenez bien soin de lire les consignes et les indications fournies dans le sujet.

Bon courage, et bonne composition.

— On rappelle qu'on peut accéder à l'adresse d'une variable `a` à l'aide de la syntaxe `&a`, et qu'on peut accéder à la variable contenue à une adresse contenue dans un pointeur `b_ptr` à l'aide de la syntaxe `*b_ptr`.
— On rappelle qu'il est possible d'accéder à l'élément d'indice `k` dans un tableau `t` à l'aide de la syntaxe `t[k]`.
— On rappelle que la fonction `sizeof` renvoie la taille d'une donnée ou d'un type en octet, que la fonction `malloc` alloue une zone mémoire sur le tas d'une taille donnée en entrée et renvoie si possible un pointeur vers cette zone, et que la fonction `free` libère une zone allouée sur le tas par malloc à l'aide du pointeur qu'elle avait renvoyée.
— Il est possible de définir une structure en C avec la syntaxe suivante :

```
struct nom_de_la_struct {
    type1 champs1 ;
    type2 champs2 ;
    ...
    typen champsn;
};
```

— Par la suite, on peut accéder et modifier un champs de ce type à l'aide des syntaxes suivantes :

```
variable.champs1
```

```
variable.champs1 = ...;
```

— On peut créer un nouveau type à l'aide d'un autre à l'aide de la syntaxe suivant :

```
typedef type_deja_defini nouveau_type;
```

Exercice 1. Représentation petit-boutiste

On rappelle que les entiers sont représentés classiquement avec l'ordre grand-boutiste en mémoire : c'est le bit de poid fort qui se trouve en premier.

Par exemple, le tableau de 8 bits suivant représente le nombre $a_7 2^7 + a_6 2^6 + a_5 2^5 + a_4 2^4 + a_3 2^3 + a_2 2^2 + a_1 2^1 + a_0 2^0$.

a_7	a_6	a_5	a_4	a_3	a_2	a_1	a_0

On s'intéresse à la représentation petit-boutiste dans cet exercice : le bit de poid faible est le premier représenté dans le tableau. Ainsi, le nombre précédent se représente de la manière suivante :

a_0	a_1	a_2	a_3	a_4	a_5	a_6	a_7

Par exemple, on peut représenter le nombre $11 = \overline{1011}^2$ de la manière suivante :

1	1	0	1	0	0	0	0

Tous les nombres sont considérés positifs dans la suite de cet exercice.

1. Sur 8bits, comment représenter le nombre 87 en représentation petit-boutiste ?
2. À quel nombre correspond la représentation petit-boutiste suivante ?

0	0	0	1	1	0	0	1

Dans la suite, on considère que les entiers sont représentés sur 4 octets, c'est-à-dire sur 32 bits. On se donne donc un type pour les entiers petit-boutiste :

```c
typedef unsigned int petitbout;
```

Le type est le même que pour des entiers non signés, mais l'interprétentation n'est pas la même.

3. (a) Proposer une fonction de prototype `petitbout vers_petitboutiste(unsigned int n)` qui transforme un entier n de sa représentation grand-boutiste vers sa représentation petit-boutiste.
   (b) Comment peut-on implémenter une fonction qui fait le sens inverse à partir de la fonction précédente ?

   Dans la suite on n'utilisera pas le changement de représentation pour implémenter les opérations : on opérera directement sur les représentation petit-boutiste au lieu de repasser par les représentations grand boutiste. Ainsi, il n'est pas possible d'implémenter la fonction suivante à l'aide du code :

```c
bool est_plus_grand(petitbout a, petitbout b){
    return vers_grandboutiste(a) >= vers_grandboutiste(b);
}
```

4. Proposer une fonction de prototype `bool est_plus_grand(petitbout a, petitbout b)` qui renvoie si le premier argument est plus grand que le second dans la représentation petit-boutiste.
5. Proposer une fonction de prototype `petitbout ajouter(petitbout a, petitbout b)` qui renvoie la somme de deux entiers dans leurs représentations petit-boutistes.
6. Proposer une fonction de prototype `petitbout multiplier(petitbout a, petitbout b)` qui renvoie le produit de deux entiers dans leurs représentations petit-boutistes.

Correction :

1. On a :

$$87 = 2^0 + 2^1 + 2^2 + 2^4 + 2^6$$

   Donc, on peut obtient :

1	1	1	0	1	0	1	0

2. On calcule :

$$2^3 + 2^4 + 2^7 = 152$$

   Cette représentation petit-boutiste correspond donc à l'entier $\boxed{152}$.

3. (a) On suppose que les entiers sont représentés sur 4 octets. On calcule la réponse en appliquant des masques successifs : si le bit à l'indice i vaut 1 dans l'entrée, alors le bit à l'indice $31-i$ vaut 1.

```c
petitbout vers_petitboutiste(unsigned int n){
    petitbout resultat = 0;
    for (int i = 0; i<32; i++){
        if ((n &(1<<i))!=0){
            resultat|=(1<<(31-i));
        }
    }
    return resultat;
}
```

   (b) Étant donné que l'inverse de l'ordre petit-boutiste est l'ordre grand-boutiste, et inversement, on peut utiliser le même procédé que dans la fonction précédente.

   Cependant, il faut changer le prototype de sorte à bien utiliser les bon types.

   On peut donc réutiliser la fonction précédente quitte à faire des conversion avant et après son application.

4. On parcours les bits en commençant par le bit le plus à droite (celui de plus grande importance) en concluant dès que possible.

   En cas d'égalité, on renvoie vrai : il s'agit d'une inégalité large.

```
bool est_plus_grand(petitbout a, petitbout b){
    for (int i = 0; i<32; i++){
        if ((a &(1<<i))>(b &(1<<i))){
            return true;
        }
        if ((a &(1<<i))<(b &(1<<i))){
            return false;
        }
    }
    return true;
}
```

5. On utilise un calcule des retenues de la même manière que dans le DM : on calcul à chaque fois un nombre qui correspond à l'ensemble des retenues.

   Attention : au concours, il faudra justifier le procédé (preuve de terminaison et de correction), ou utiliser le procédé plus simple de l'ajout classique.

```
petitbout ajouter(petitbout a, petitbout b){
    petitbout retenue = 0;
    while (b!=0){
        retenue = (a&b)>>1;
        a = a^b;
        b = retenue;
    }
    return a;
}
```

6. On utilise le même procédé que la multiplication telle qu'elle est enseignée, on fait des additions de la multiplication de b avec chacun des chiffres de a.

   Comme pour chaque chiffre, il y a soit 0 soit 1, on sait que l'on aura besoin soit de ne rien ajouter, soit d'ajouter b décalé d'un nombre de bits qui dépend de la position du chiffre.

```
petitbout multiplier(petitbout a, petitbout b){
    petitbout resultat = 0;
    for (int i = 0; i<32; i++){
        if ((n &(1<<(31-i)))!=0){
            resultat = ajouter(resultat, b>>i);
        }
    }
    return resultat;
}
```

Exercice 2. Achat et revente

Soit $n \geqslant 1$. On se donne un tableau de n entiers de a_0 à a_{n-1}.

a_0	a_1	$\cdots$	a_{n-1}

L'objectif est de calculer la grandeur suivante :

$$S = \max_{0 \leqslant i \leqslant j \leqslant n-1} (a_j - a_i)$$

Attention, la contrainte $i \leqslant j$ est importante.

1. Dans cette question uniquement, on suppose que le tableau est le tableau suivant :

7	0	4	-1

Combien vaut S pour le tableau ?

2. Construire un tableau de taille au moins $n = 2$ tel que $S = 0$.
3. Proposer une fonction naïve de prototype `int s(int * tableau, int taille)` qui calcule S à partir du tableau et de sa taille.

   La complexité temporelle devra être en $O(n^2)$, et on le justifiera brièvement.
4. On pose les grandeurs suivantes :

$$m_k = \min_{0 \leqslant i \leqslant k} a_i$$
$$s_k = \max_{0 \leqslant i \leqslant j \leqslant k} (a_j - a_i)$$

   (a) Combien valent m_0 et s_0 ?
   (b) Proposer une formule de récurrence pour calculer m_{k+1} à partir de m_k et a_{k+1}.
   (c) Proposer une formule de récurrence pour calculer s_{k+1} à partir de s_k, m_k et a_{k+1}.
   (d) En déduire une fonction en $O(n)$ pour calculer S.

   La complexité de cet algorithme sera brièvement justifiée.
   (e) Montrer la terminaison et la correction de cette fonction.
5. — Proposer une fonction qui calcule i_0 et j_0 tels que $i_0 \leqslant j_0$ et $a_{j_0} - a_{i_0} = S$.

   La réponse de la fonction sera renvoyée au travers de deux pointeurs passés en arguments, et son prototype sera le suivant :

```c
void calculer_ijzero(int * tableau, int taille, int * izero_ptr, int * jzero_ptr)
```

   Au terme de l'exécution de la fonction, la valeur contenue dans la case mémoire pointée par `izero_ptr` contiendra i_0 tandis que celle pointée par `jzero_ptr` contiendra j_0.
6. On cherche désormais à renvoyer une structure avec notre fonction qui contienne à la fois i_0 et j_0.
   (a) Proposer la définition d'une structure `struct reponse_ijzero` qui puisse contenir les entiers i_0 et j_0 en C.
   (b) Proposer une fonction de prototype `struct reponse_ijzero renvoyer_ijzero(int * tableau, int taille)` qui utilise cette structure pour définir une fonction qui renvoie à la fois i_0 et j_0.

   On réutilisera la fonction `calculer_ijzero`.

Correction :

1. Il y a 10 choix de i et j possibles. Le maximum est atteint pour $i = 1$ et $j = 2$, on trouve alors : $\boxed{S = 4}$
2. On prend un exemple de tableau dont les éléments sont par ordre décroissant (au sens large) :

0	0

3. On propose une fonction qui teste toutes les possibilités pour $0 \leqslant i \leqslant jn$.

```c
int s(int * tableau, int taille){
    int s_actuel = 0;
    for (int i = 0; i<taille; i++){
        for (int j = i; j<taille; j++){
            int nouveau = tableau[j] - tableau[i];
            if (nouveau>s_actuel){
                s_actuel = nouveau
            }
        }
    }
    return s_actuel;
}
```

Sa complexité temporelle est bien en $O(n^2)$ car on parcourt les $\frac{n(n+1)}{2}$ possibilités pour i et j, et qu'on réalise un traitement en $O(1)$ pour chacune de ces possibilités.

4. (a) D'une part, $m_0 = \min_{0 \leqslant i \leqslant 0} a_0 = a_0$, d'autre part $s_0 = \max_{0 \leqslant i \leqslant j0}(a_j - a_0) = a_0 - a_0 = 0$.

   Donc $\boxed{m_0 = a_0 \text{ et } s_0 = 0.}$

   (b) Le minimum des $k+1$ éléments est le minimum du minimum des k premiers élément et du $k+1$-ième élément.

$$\begin{aligned} m_{k+1} &= \min_{0 \leqslant i \leqslant k+1} a_i \\ &= \min(\min_{0 \leqslant i \leqslant k} a_i, a_{k+1}) \\ &= \min(m_k, a_{k+1}) \end{aligned}$$

   Ainsi, on a $\boxed{m_{k+1} = \min(m_k, a_{k+1}) \text{ pour tout } k \geqslant 0.}$

   (c) Le maximum de $(a_j - a_i)$ pour $j \leqslant k+1$ est atteint pour $j \leqslant k$ ou pour $j = k+1$. Dans le premier cas, ce maximum est égal à s_k, dans le second cas, il est atteint pour $i \leqslant k$ égal à l'indice d'un minimum de des a_i.

$$\begin{aligned} s_{k+1} &= \max_{0 \leqslant i \leqslant j \leqslant k+1} (a_j - a_i) \\ &= \max\left(\max_{0 \leqslant i \leqslant j \leqslant k} (a_j - a_i), \max_{0 \leqslant i \leqslant k+1} (a_{k+1} - a_i)\right) \\ &= \max\left(s_k, a_{k+1} - m_{k+1}\right) \end{aligned}$$

   Or, $s_{k+1} \geqslant 0$, donc, on peut écrire, pour tout $k \geqslant 0$, $\boxed{s_{k+1} = \max\left(s_k, a_{k+1} - m_k\right)}$

   (d) On mets à jours s_k et m_k au fur et à mesure de sorte à pouvoir calculer S :

```c
int s(int * tableau, int taille){
    int s_k = 0;
    int m_k = tableau[0];
    for (int i = 1; i<taille; i++){
        int val = tableau[i] - m_k;
        if (tableau[i]<m_k){
            m_k = tableau[i];
        }
        if (val> s_k){
            s_k = val;
    }
    return s_k;
}
```

   Il s'avère que l'ordre dans lequel on calcule la nouvelle valeur pour m_k et s_k n'est pas importante car s_0 est égal à 0.

   (e) Dans notre fonction i est une variable entière qui augmente de 1 exactement à chaque tours de boucle et qui est majorée par la taille : la boucle for termine, et $\boxed{\text{la fonction termine donc.}}$

   Par ailleurs, lorsque la fonction termine, on a $k = n-1$, et on a donc s_k égal à s_{n-1}. Or $s_{n-1} = S$ par définition, donc notre fonction renvoie bien la grandeur attendue. $\boxed{\text{Elle est correcte.}}$

Problème : Gestion de la mémoire sur le tas

L'objectif de ce problème est d'implémenter une fonction d'allocation dans une zone mémoire. L'idée est d'utiliser une grande zone contiguë en mémoire et d'organiser les différentes régions de la mémoire à l'aide de fonctions sans utiliser malloc et free.

Partie I : Allocation d'une mémoire

On décide de représenter une adresse par le type suivant :

```
typedef uint16_t adresse_t;
```

On rappelle que le type uint16_t correspond à un entier non signé sur 16 bits.

Question 1. Combien d'adresse différentes peut on avoir avec ce type ?

Correction : Il y a $\boxed{2^{16} \text{ adresses possibles}}$, car un entier non signé sur N bits peut donner des valeurs de 0 à $2^{16} - 1$.

On pourra considérer que ce nombre d'adresse est enregistré dans une constante du nom de taille_memoire .

On représente la mémoire par un tableau d'entiers de type int dont le type est :

```
typedef int * memoire_t
```

Question 2. Pourquoi avons-nous utilisé ce type là pour représenter le tableau qui représente la mémoire ?

Correction : Nous cherchons à représenter un tableau d'entiers. Les tableaux peuvent être manipulés avec des pointeurs vers le premier élément en C.
Donc nous pouvons utiliser un $\boxed{\text{pointeur d'entier pour représenter la mémoire}}$.

La taille est fixée à la taille maximale donnée par les adresses possibles, c'est-à-dire à la valeur de taille_memoire.

Question 3. Proposer une fonction de prototype memoire_t allouer_memoire_totale() qui renvoie un pointeur vers le premier élément d'un tableau alloué sur le tas de cette taille là.
On prendra soin de vérifier que l'allocation mémoire a réussi, et on utilisera la fonction abort dans le cas contraire. Cette fonction s'appelle sans argument et n'a pas de valeur de retour.

Correction : On alloue la mémoire sous la forme d'un tableau sur le tas. On utilise la constnate taille_memoire pour le nombre d'élément.

```
memoire_t allouer_memoire_totale(){
    memoire_t mem = (memoire_t) malloc (taille_memoire * sizeof(int));
    if (mem == NULL){
        abort();
    }
    return mem;
}
```

Question 4. Si un entier est représenté sur 32 bits, quelle est la taille de ce tableau en mémoire en nombre de bits ?

Correction : La mémoire est un tableau de 2^{16} cases de 32 bits. Le total est donc de $2^{16} \times 32$ bits, et donc de $\boxed{2097152}$.
(2^{21} était accepté comme réponse.)

On a bien alloué ce tableau sur le tas : on ne fera pas de nouvelles allocations sur le tas, mais on simulera les allocation mémoire dans cette mémoire.

Ainsi, quand on parlera de mémoire de type `memoire_t` ici, on parlera de la mémoire simulée par le tableau, et quand on parlera d'adresse de type `adresse_t` on fera référence aux adresses dans cette mémoire.

Question 5. Proposer une fonction de prototype `int acceder(memoire_t mem, adresse_t adr)` qui renvoie la valeur donnée dans la mémoire `mem` à l'adresse `adr`.

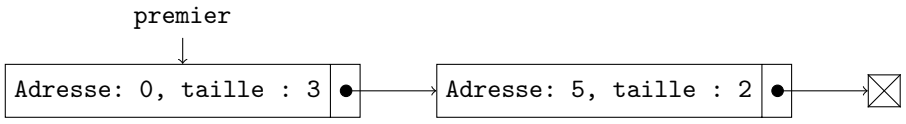
Correction : On utilise le fait qu'on peut manipuler `mem` comme un tableau :

```c
int acceder(memoire_t mem, adresse_t adr){
    return mem[adr];
}
```

Partie II : Listes chaînées des espaces libres

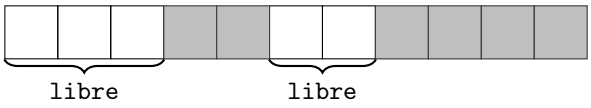
Pour pouvoir gérer les espaces libres dans cette mémoire, on utilisera une liste chaînée des blocs libres dans la mémoire.

Par exemple, avec la liste chaînée suivante :



On décrit l'existence de deux blocs libres dans la mémoire, un qui commence en 0 de taille 3, et un deuxième qui commence en 5 de taille 2.

Ainsi, la mémoire pourrait ressembler à ceci :



On pourrait représenter cette liste chaînée à l'aide de la structure suivante :

```c
typedef struct _bloc {
    adresse_t debut;
    uint16_t taille;
    stuct _bloc * suivant;
} bloc_t;
```

Cependant, le problème d'utiliser une liste chaînée de ce type est que nous devons allouer les cellules sur le tas au lieu d'utiliser notre mémoire, et nous ne voulons justement pas faire d'allocations sur le tas pour chacune des cellules.

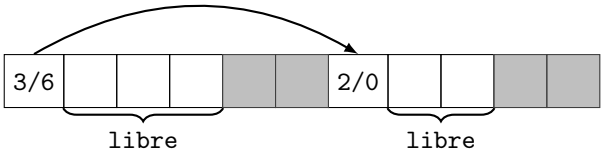
Au lieu d'utiliser la structure précédente, nous allons donc stocker directement les informations sur les blocs dans la mémoire. En effet, une adresse (au sens du type `adresse_t`) et la taille d'un bloc sont toutes les deux représentés sur 16bits, et on peut donc utiliser une case mémoire du tableau d'entiers (qui sont codés sur 32 bits) pour encoder les deux.

Certaines cases sont donc des cases d'entête de blocs qui contiennent les informations liés au bloc. Chaque case mémoire d'entête contient la taille du bloc mémoire libre qui la suit directement, et l'adresse (de type `adresse_t`) du prochain entête.

On impose par ailleurs que la première case du tableau est toujours une case d'entête, et que l'entête du dernier bloc libre pointe vers la première case du tableau.

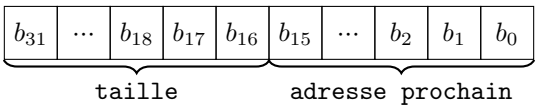
De plus, on impose que les adresses des différents entêtes sont toujours **croissante** lorsqu'on réalise un parcours de la liste chaînée en partant de la première case, sauf pour le dernier bloc (qui doit pointer vers la première case du tableau).

Dans l'exemple suivant, les cases d'entête sont décrite au format taille/adresse du prochain entête.



On remarque qu'on doit utiliser plus de cases en mémoire à cause de la présence de cases d'entête.

Dans une case d'entête, on suppose que les 16 premiers bits sont réservés à la taille du bloc libre, et les 16 suivants sont réservé à l'adresse du prochain entête.



Ainsi, dans l'exemple précédent, la taille du bloc libre est $\overline{b_{31}\cdots b_{16}}^2$, tandis que l'adresse du prochain entête est $\overline{b_{15}\cdots b_0}^2$.

Question 6. Proposer une fonction de prototype `uint16_t obtenir_taille(int n)` qui obtient la taille du bloc mémoire libre à partir de la valeur n contenue dans la case d'entête.

Correction : La taille correspond aux 16 premiers bits de poid fort. On doit donc décaler vers la droite les bits.

On réalise la conversion explicitement, mais cela n'est pas nécessaire.

```
uint16_t obtenir_taille(int n){
    return (uint16_t) (n>>16);
}
```

Question 7. Proposer une fonction de prototype `adresse_t obtenir_adresse(int n)` qui obtient l'adresse du prochain entête à l'aide de la valeur n contenue dans la case d'entête.

Correction : Il suffit de réaliser la conversion vers le type attendu :

```
adresse_t obtenir_adresse(int n){
    return (adresse_t) n;
}
```

Question 8. Proposer une fonction de prototype `int calculer_entete(uint16_t taille, adresse_t a)` qui à partir d'une taille de bloc mémoire et d'une adresse du prochain entête, calcule le contenu de l'entête.

Correction : On pourrait utiliser les conversions implicites des opérateurs de décalages pour se passer de variables locales, mais on préfère faire les conversions explicitement.

```
int calculer_entete(uint16_t taille, adresse_t a){
    int taille32 = ((int) taille)<<16;
    int a32 = (int) a;
    return taille32 + a32;
```

Question 9. Quel est le contenu de la première case du tableau si l'entièreté de la mémoire est libre sous la forme d'un seul bloc mémoire ?

Correction : Si l'entièreté de la mémoire est constituée d'un seul bloc libre, l'adresse du prochain bloc est 0, et la taille est de taille_memoire -1, c'est-à-dire $2^{16}-1$.

Donc la valeur dans la première case sera $\boxed{2^{32}-2^{16}}$, c'est-à-dire 16 bits qui valent 1, puis 16 bits qui valent 0.

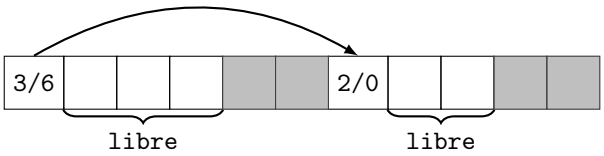
Question 10. En déduire une fonction une fonction `void initialiser(memoire_t mem)` qui initialise la mémoire pour être entièrement libre sous la forme d'un seul bloc mémoire libre (sauf la première case qui doit être un entête).

Correction : On utilise la question précédente et le fait qu'on peut utiliser les dépassement mémoire pour obtenir le résultat espéré.

```
void initialiser(memoire_t mem){
    meme[0] = -(1<<16);
}
```

Question 11. Proposer une fonction de prototype `uint16_t taille_libre_totale(memoire_t mem)` qui détermine la place mémoire libre totale, c'est-à-dire la somme de la taille des blocs mémoires libres (sans compter les entêtes).

Par exemple, dans l'exemple suivant, le nombre de cases libre est 5.



Correction : On réalise un parcours de la liste des blocs mémoire libre. La variable actuelle contient l'adresse actuelle de la cellule que l'on étudie. On utilise un `do ... while (...)` ; de sorte à pouvoir réaliser au moins une fois la boucle.

```
uint16_t taille_libre_totale(memoire_t mem){
    uint16_t resultat = 0;
    adresse_t adr = 0;
    do{
        resultat += obtenir_taille(mem, adr);
        adr = obtenir_adresse(mem, adr);
    } while (adr!=0);
    return adr;
}
```

Attention, le point-virgule est nécessaire dans une syntaxe `do .... while (...) ;`

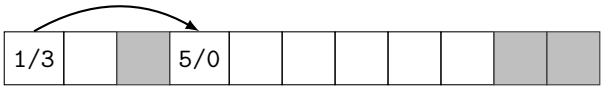
Question 12. Quelle est la complexité temporelle asymptotique de la fonction `taille_libre_totale` en fonction de N le nombre de blocs mémoire libre ?

Correction : On parcours chacun des blocs une fois, et on réalise des opérations en temps constant pour chaque bloc : la complexité totale est donc en $O(N)$.

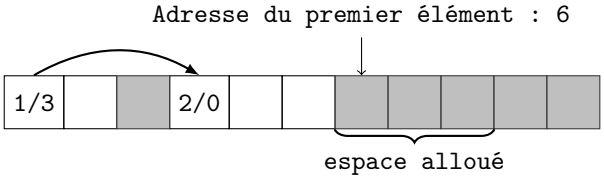
Partie III : Allocation et libération

Pour allouer une case mémoire, on réalise un parcours des blocs libres, et on prend le premier dont la taille est suffisante pour accueillir les données.

Par exemple, en considérant la mémoire suivante :



Si on essaye d'allouer une zone de taille 3, on ne peut pas l'allouer dans le premier bloc libre (il n'y a pas la place), et on doit donc aller au suivant.



Ici, on a modifé seulement la taille de la case mémoire, et on a renvoyé l'adresse mémoire correspondant au premier élément du bloc alloué : 6.

On suppose qu'on n'a jamais à modifier le chaînage lui-même, par exemple, si on veut allouer une zone de taille 3 à nouveau dans l'exemple précédent, il ne sera pas possible de le faire, car le deuxième bloc libre n'est pas de taille suffisante à cause de la présence de l'entête.

Dans le cas où l'allocation échoue, on renvoie 0 : étant donné que la première case est nécessairement un bloc d'entête, on ne pourra jamais légitemement renvoyer 0 pour une allocation réussie.

Question 13. Proposer une fonction d'allocation qui cherche le premier bloc libre dont la taille est suffisante pour accueillir la zone mémoire que l'on cherche à allouer, et qui le modifie de sorte à pouvoir renvoyer le première élément de la zone mémoire allouée.

La fonction aura le prototype suivant :

```
adresse_t allouer(uint16_t taille, memoire_t mem)
```

Correction : On parcourt le tableau jusqu'à trouver une case qui est vide. Dans le cas où on ne trouve aucun espace mémoire, on renvoie 0 qui ne peut pas être un début d'adresse alloué, et qui peut donc être réservé pour une erreur :

```
adresse_t allouer(uint16_t taille, memoire_t mem){
    adresse_t actuel = 0;
    do{
        uint16_t ancienne_taille = obtenir_taille(mem, actuel);
        if (ancienne_taille >= taille){
            mem[actuel] = calculer_entete(ancienne_taille - taille,
    obtenir_adresse(mem, actuel));
            return actuel + ancienne_taille - taille + 1;
        }
        actuel = obtenir_adresse actuel;
    } while (actuel != 0);
    return 0;
}
```

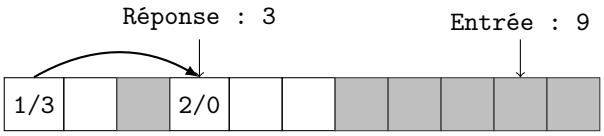
Question 14. Que peut-on dire du nombre N de blocs de mémoire libre à chaque appel à allouer ?

Correction : Étant donné que l'on ne modifie pas les entêtes, le nombre d'entêtes et le nombre de blocs libres n'est donc pas modifié, N est donc constant avec chaque appel à allouer.

Avant de pouvoir réaliser la libération d'une zone de la mémoire, on a besoin d'une fonction supplémentaire.

Question 15. On cherche à calculer l'adresse de l'entête la plus grande qui soit strictement plus petite que l'adresse d'un bloc alloué en mémoire.

Par exemple, avec l'entrée 9 dans la mémoire suivante, on doit obtenir 3 :



1. Pourquoi une telle adresse existe toujours ?
2. Proposer une fonction de prototype `adresse_t entete_precedente(adresse_t a, memoire_t mem)` qui renvoie l'adresse de l'entête la plus grande de sorte à ce que l'adresse de sortie soit inférieure strictement à l'adresse d'entrée, qui est supposée être l'adresse d'une case allouée en mémoire.
3. Quelle est la complexité dans le meilleur et le pire des cas de cette fonction en fonction de N le nombre de bloc mémoire libre ?

Correction :

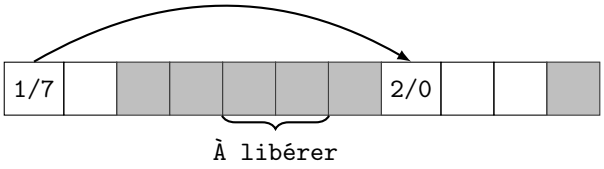
1. Il existe au moins une adresse inférieures strictement, celle en 0. L'ensemble des adresses inférieures est donc un ensemble d'entiers borné et majoré, une telle adresse existe toujours.
2. On prend garde au cas où la dernière entête est l'entête qui nous intéresse.

```
adresse_t entete_precedente(adresse_t a, memoire_t mem){
    adresse_t actuel = 0;
    while ((obtenir_suivant(mem, actuel) < a )&&(obtenir_suivant(mem, actuel)!=
    0)){
        actuel = obtenir_suivant(actuel);
    }
    return actuel;
}
```

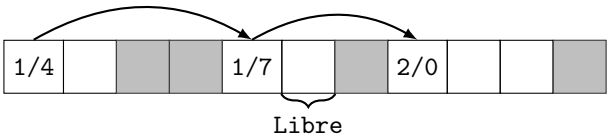
3. Dans le meilleur des cas, on trouve l'entête dès le début, et on n'exécute pas la boule pour un total en $O(1)$.

   Dans le pire des cas, on doit parcourir toutes les cellules avec des opérations en $O(1)$ pour chaque cellule, pour un total de $O(N)$

On cherche maintenant à réaliser la libération d'une zone mémoire. L'idée est la suivante : on rajoute le bloc mémoire libéré dans la liste. Par exemple, sur le tableau suivant, en libérant la zone de taille 2 qui commence à l'indice 4, il est nécessaire de rajouter un bloc de mémoire libre dans notre liste chaînée en utilisant le premier élément de la zone libérée en tant qu'entête.



On obtient donc :



Question 16. Proposer une fonction de libération de la mémoire qui prend en argument l'indice du premier élément de la zone à libérer, et la taille de la zone à libérer. La taille de la zone mémoire à libérer est non nulle.

Le prototype de la fonction attendue est la suivante :

```
void liberer(adresse_t debut, uint16 taille, memoire_t mem)
```

On pourra supposer que l'entrée est valide : c'est-à-dire qu'elle correspond bien à une zone mémoire déjà allouée de taille au moins $n > 0$.

Correction : On réutilise la fonction précédente pour savoir quelle entête modifier.

```
void liberer(adresse_t debut, uint16_t taille, memoire_t mem){
    adresse_t precedente = entete_precedente(mem, debut);
    adresse_t suivante = obtenir_entete(mem, precedente);
    mem[precedente] = calculer_entete(debut, obtenir_taille(precedente));
    mem[debut] = calculer_entete(suivante, taille - 1);
}
```

Question 17. Que peut-on dire du nombre de bloc mémoire libre N à chaque appel à liberer ?

Correction :
Le nombre de blocs mémoire libre augmente de 1 exactement à chaque appel à libérer.

Question 18 (⋆). Ici, on a supposé qu'il était nécessaire d'avoir la taille de la zone mémoire pour réaliser la libération. Comment pourrait-on modifier notre code pour ne pas avoir besoin de cette information ?

Correction : Pistes de solution :
- Utilisation d'une autre liste chaînée ;
- utilisation d'un tableau dynamique avec les adresses et les tailles dans une zone réservée à la fin de la mémoire.

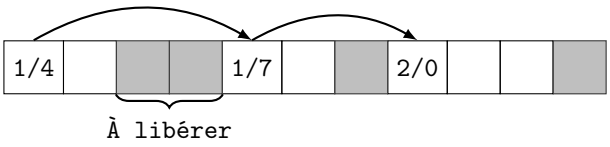
Partie IV : Défragmentation

Le problème de cette stratégie est qu'on peut arriver à une *fragmentation de la mémoire*. Dans notre cas, cela signifie que notre mémoire libre est découpée en de nombreux morceaux, ce qui a un impact sur les performances pour l'allocation et la place restante.

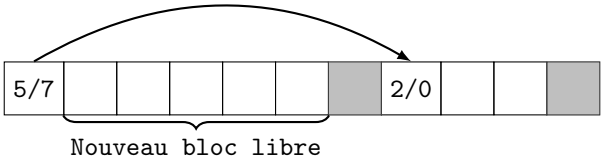
Question 19. Que se passe-t-il si on alloue taille_memoire − 1 zones de taille 1, puis qu'on les libère toutes ? Quel est le problème ?

Une solution est de faire un peu plus lors de la libération : on vérifie s'il y a avant ou après un bloc de mémoire libre que l'on puisse libérer.

Par exemple, en partant de la configuration suivante, si l'on l'on cherche à libérer la zone qui commence à l'indice 2 et de taille 2 :



On est sensé obtenir le résultat suivant :

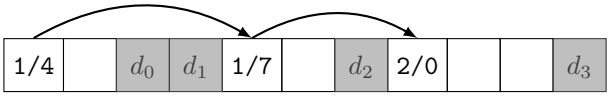


Question 20. Proposer une fonction de prototype void liberer_et_fusionner(adresse_t debut, uint16 taille , memoire_t mem) qui libère un espace en mémoire donné par son adresse de début et la taille de la zone à libérer, et qui fusionne si possible les cellules.

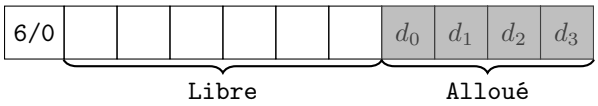
Question 21. Montrer sur un exemple d'allocations et de libérations successives qu'il est possible d'avoir un nombre de blocs libres en $O(T)$ où T est la taille totale du tableau en nombre de cases, c'est-à-dire qu'il existe $C > 0$ une constante réelle telle que pour tout T, il soit possible d'avoir au moins CT blocs libres.

Pour résoudre ce problème, on peut décider de déplacer tous les blocs libres au début de la mémoire. Si la valeur dans les blocs libres de la mémoire n'a pas d'importance, il est tout de même nécessaire de déplacer les blcos qui ont été alloués pour le moment.

Par exemple, en se donnant la situation suivante :



On est sensé obtenir la situation suivante :



Question 22 (⋆). Proposer une fonction qui défragmente la mémoire libre dans son ensemble de sorte à ce qu'il n'y ait plus qu'un seul bloc mémoire libre.

On s'assurera que les données qui avaient été allouées sont recopiées à la fin de la mémoire, et qu'elles sont dans le même ordre d'apparition qu'initialement.

Le prototype attendu est le suivant :

```
1 void defragmenter(memoire_t mem)
```

On prendra soin d'avoir une complexité temporelle en $O(T)$ où T est la taille de la mémoire. On limitera l'usage de mémoire supplémentaire par rapport à l'entrée. La complexité mémoire sera annoncée sans justification.

Correction : Indiction : il faut déplacer les éléments occupés à gauche d'un bloc libre vers sa droite en partant du dernier bloc libre puis en itérant pour garder une complexité en $O(T)$. Il est possible d'utiliser une fonction récursive pour avoir une complexité spatiale en $O(N)$, mais il est sinon possible d'utiliser les entêtes elle-même pour se souvenir de comment remonter dans le tableau.

Il est par ailleurs possible de recopier le tableau brutalement dans un autre tableau, mais le coût en mémoire est alors en $O(T)$. Cela est cependant plus facile à implémenter.

Le problème de cette stratégie, est qu'en déplaçant la mémoire réservée par les différents appels à `allouer`, on perd l'intérêt principal de l'allocation mémoire qui est justement qu'on ne touche pas aux zones allouées dans la mémoire. Les adresses données par les différents appels à `allouer` ne sont donc plus valides même si les données sont quelque part en mémoire.

Question 23 (⋆). Proposer une solution pour pouvoir tout de même relocaliser les espaces alloués dans la mémoire sans que les informations données par `allouer` soient affectés par la défragmentation.

On pourra réserver une partie de la mémoire pour stocker d'autres informations.

Correction : Indication : il est possible d'utiliser une grille de correspondance entre les adresses utilisées depuis l'extérieur, et les adresses effectives en mémoire.