SOMME D'UNE PARTIE

Le problème dit SubsetSum est le suivant :

Entrées : un (multi-)ensemble $X = x_0, \dots, x_{n-1}$ d'entiers naturels et un entier naturel S ;

Sortie : `true` s'il existe une partie $Y \subset X$ telle que $\sum_{x \in Y} x = S$, `false` sinon.

Remarques

- On peut bien sûr vouloir obtenir un exemple de partie Y convenable quand il en existe une.
- Dans tout le sujet, ensemble signifiera en fait multi-ensemble : cela signifie simplement qu'il peut y avoir des répétitions dans les éléments donnés en entrée (on pourrait exiger que ce ne soit pas le cas) et qu'une même somme peut *a priori* être obtenue de plusieurs manières.

Dans tout le sujet, on représentera les entiers x_i par des `uint64_t` et l'on supposera systématiquement qu'il n'y a pas de problème de débordement. Pour alléger, on définit :

```
typedef uint64_t T;
```

1 Solution en force brute

Exercice L.1

p. 6

1. Écrire une fonction `naive_decision` ayant la spécification suivante :

   Prototype :

   ```
   bool naive_decision(T arr[], int len, T goal);
   ```

   Entrées :

   - `arr` pointe vers un bloc de longueur `len` contenant les entiers $x_0, \dots, x_{\texttt{len}-1}$;
   - `goal` est la somme S à atteindre.

   Sortie : `true` s'il existe un sous-ensemble des éléments de `arr` dont la somme vaut `goal`, `false` sinon.

   Remarque

   Cette fonction devrait être extrêmement simple ; en particulier, il n'y a pas besoin de fonction auxiliaire.

2. Déterminer la complexité en temps et en espace de la fonction `naive_decision`. Donner un ordre de grandeur de la valeur maximale de `len` que l'on peut traiter en un temps raisonnable (de l'ordre de la seconde ou de la minute).

Exercice L.2

p. 6

1. Écrire une fonction `naive_solution_aux` ayant la spécification suivante :

   Prototype :

   ```
   bool naive_solution_aux(T arr[], int len, T goal, bool sol[]);
   ```

Entrées :

- `arr` et `sol` sont deux tableaux de longueur `len` ;
- `goal` est la somme à atteindre.

Sortie : un booléen indiquant si la somme peut être atteinte.

Effets secondaires : si la valeur de retour est `true`, alors `sol` code une solution. Plus précisément, on a alors la somme des `arr[i]` pour les i tels que `sol[i]` soit vrai qui vaut `goal`. Si la valeur de retour est `false`, alors le contenu de `sol` est sans importance.

2. Écrire une fonction `naive_solution` qui renvoie :
   - un pointeur vers un bloc alloué (de longueur `len`) codant une solution, s'il en existe une ;
   - le pointeur `NULL` sinon.

```
bool *naive_solution(T arr[], int len, T goal);
```

Exercice L.3 p. 7

Le fichier fourni contient deux fonctions :

- `read_elements` permet de lire une instance du problème depuis un fichier. Le format attendu est `n S x0 x1 ... x_(n - 1)` et la fonction a le comportement suivant :
  — elle renvoie un pointeur vers un bloc alloué contenant $x_0, \dots, x_{n-1}$;
  — elle modifie les valeurs pointées par `len` et `goal` pour qu'elles correspondent respectivement à n et à S.

```
T *read_elements(FILE *fp, int *len, uint64_t *goal);
```

- `print_solution` permet l'affichage d'une solution.

```
void print_solution(FILE *fp, uint64_t elements[], int len, bool solution[]);
```

Écrire un programme ayant le comportement suivant :

- si aucun argument n'est passé en ligne de commande, il lit une instance sur l'entrée standard et écrit le résultat sur la sortie standard ;
- si un unique argument est passé en ligne de commande, cet argument est considéré comme un nom de fichier et l'instance est lue depuis ce fichier – le résultat est toujours écrit sur la sortie standard ;
- si deux arguments sont passés, le premier est utilisé comme fichier d'entrée et le second comme fichier de sortie.

Dans tous les cas, le résultat sera une ligne contenant `Yes` ou `No` suivant qu'une solution existe ou non, suivie le cas échéant de la solution.

2 Meet in the middle

La technique dite *Meet in the middle* permet d'accélérer certains algorithmes de complexité exponentielle. Elle consiste à résoudre deux problèmes de taille $n/2$ et à en déduire le résultat pour notre problème de taille n. La différence avec le *diviser pour régner* « classique » est que l'on ne résoudra pas exactement le même problème, et qu'il sera donc impossible d'étendre la méthode récursivement. Ici, le principe est le suivant :

- on divise notre ensemble $X = x_0, \dots, x_{n-1}$ en deux parties $A = x_0, \dots, x_{\lfloor n/2 \rfloor - 1}$ et $B = x_{\lfloor n/2 \rfloor}, \dots, x_{n-1}$;

- on calcule $s(A)$ (respectivement $s(B)$) l'ensemble des sommes que l'on peut obtenir en choisissant des éléments de A (respectivement de B) :

$$s(A) = \left\{ \sum_{x \in Y} x \mid Y \subset A \right\}$$

- à partir de $s(A)$ et $s(B)$, on détermine si $S \in s(X)$ (qui est notre question initiale).

Représentation d'une partie On suppose dans toute cette partie que $n \leqslant 64$, ce qui n'est pas vraiment une restriction vu la complexité des algorithmes que nous allons manipuler. Par souci de clarté, on définit :

```
typedef uint64_t set_t;
```

Si s est de type `set_t`, il représente la partie de $[0..63]$ correspondant à ses bits valant 1 : par exemple, l'entier $21 = 2^0 + 2^2 + 2^4$ représente la partie $\{0, 2, 4\}$.

Représentation des ensembles de sommes Pour les ensembles $s(A)$ et $s(B)$, on utilisera des tableaux indexés par les parties de A (ou de B) suivant le principe suivant : la case d'indice x contiendra la somme de la partie représentée par le `set_t` x.

Exercice L.4 p. 8

Écrire une fonction `compute_sums` ayant la spécification suivante :

Prototype :

```
void compute_sums(T arr[], set_t set, int i, T sum, T *sums);
```

Entrées :

- `arr` est un tableau de taille n (pour un certain n) représentant un ensemble A ;
- `sums` un tableau de taille 2^n destiné à recevoir les sommes des parties (c'est-à-dire l'ensemble $s(A)$) ;
- i est un entier de $[0 \ldots n-1]$;
- `set` est une partie de $[i+1 \ldots n-1]$;
- `sum` est la somme des `arr[j]` pour j dans la partie représentée par `set`.

Effets secondaires : après l'appel, les cases de `sums` correspondant à des parties s avec $s =$ `set` $\cup\, s'$ et $s' \subset [0 \ldots i]$ doivent contenir les sommes correspondantes.

Exercice L.5 p. 8

1. Quelle est la complexité de la fonction `compute_sums` ?
2. Si l'on procède de la manière la plus simple possible pour déterminer si $s \in s(X)$ à partir des ensembles $s(A)$ et $s(B)$ calculés par `compute_sums`, quelle complexité obtiendra-t-on au total ? Commenter.
3. Écrire une fonction `exists_sum` ayant le comportement suivant :

   Prototype :

   ```
   bool exists_sum(T goal, T sA[], int n, T sB[], int p){
   ```

   Entrées :

   - n et p sont des entiers positifs ou nuls ;
   - `sA` et `sB` sont des tableaux de longueur respective 2^n et 2^p.

   Précondition : `sA` et `sB` sont triés par ordre croissant.

Sortie : `true` si `goal` peut s'écrire comme un élément de `sA` et de `sB`, `false` sinon.

Attention : on veut une complexité en $O(2^n + 2^p)$ (proportionnelle à la somme des longueurs des tableaux, donc).

Exercice L.6

p. 8

Il nous reste à trier les tableaux contenant les sommes. Pour changer, nous allons utiliser la fonction `qsort` qui fait partie de la bibliothèque standard. Son prototype est le suivant :

```
void qsort(void *arr, size_t count, size_t size,
           int comp(const void*, const void*));
```

Le prototype étant un peu compliqué, une explication s'impose :

- `arr` est un pointeur vers le tableau à trier ;
- le type `void*` est celui d'un pointeur générique (c'est un pointeur vers n'importe quoi) ;
- le type `size_t` est un type entier non signé suffisamment grand pour représenter la taille de n'importe quel tableau (en pratique, il est égal à `uint64_t` sur l'immense majorité des machines) ;
- l'argument `count` indique le nombre d'éléments que le tableau contient ;
- l'argument `size` indique la taille (en octets) d'un élément du tableau (indication nécessaire puisqu'on n'a pas le type de ces éléments) ;
- le dernier argument est une fonction `comp` qui prend en entrée deux `const void*` et renvoie un `int` ;
- le qualificatif `const` signifie ici que la fonction `comp` n'a pas le droit de modifier les valeurs pointées.

L'ordre de tri est spécifié par la fonction `comp` de la manière usuelle :

- si `comp(px, py) < 0`, alors `*px` sera considéré comme plus petit que `*py` ;
- si `comp(px, py) > 0`, ce sera le contraire ;
- si `comp(px, py) == 0`, alors les deux arguments sont indiscernables pour l'ordre de tri.

Pour trier un tableau d'`uint64_t` par ordre croissant, la fonction de comparaison à utiliser sera :

```
int compare_uint64(const void *a, const void *b){
  uint64_t x = *(const uint64_t*)a;
  uint64_t y = *(const uint64_t*)b;
  if (x < y) return -1;
  if (x > y) return 0;
  return 0;
}
```

1. Que fait la ligne `uint64_t x = *(const uint64_t*)a;` (étape par étape) ?
2. Pourquoi serait-il problématique de remplacer les trois dernières lignes par `return x - y` ?
3. Écrire une fonction `sort_sums` prenant en entrée un tableau de `uint64_t` (ainsi que sa longueur) et le triant en place.

```
void sort_sums(T arr[], int len);
```

Exercice L.7

p. 8

1. Écrire une fonction `decision` ayant la même spécification que `naive_decision` mais une meilleure complexité.

```
bool decision(T arr[], int len, T goal);
```

2. Donner la complexité spatiale et la complexité temporelle de cette fonction.
3. Quelle est l'étape qui domine la complexité spatiale ?
4. Sur une machine « typique », quelle valeur maximale de n peut-on espérer traiter en un temps raisonnable, et quel est le facteur limitant (temps ou espace) ? *On précisera les hypothèses de modélisation faites sur ce qu'est une machine « typique ».*

Exercice L.8

p. 8

Écrire une fonction `solution` ayant la spécification suivante :

Prototype :

```
set_t solution(T arr[], int len, T goal, bool *found);
```

Entrées :

- `arr` est un tableau de longueur `len` contenant les éléments de X, `goal` est la somme cherchée ;
- `found` est un pointeur valide, qui constitue un argument de sortie (la valeur initialement pointée n'a aucune importance).

Sortie :

- s'il existe une solution au problème, alors le `set_t` renvoyé code une telle solution ;
- s'il n'existe pas de solution, la valeur de retour n'a aucune importance.

Effets secondaires : après l'appel, la valeur pointée par `found` indique si une solution a été trouvée.

Remarque
Il y a un certain travail à faire : on pourra être amené à définir un nouveau type, à modifier un certain nombre de fonctions. . .

Exercice L.9

p. 9

Proposer une version de `solution` ayant une complexité temporelle en $O(2^{n/2})$, et l'implémenter.

Solutions

Correction de l'exercice L.1 page 1

1.

```c
bool naive_decision(T arr[], int len, T goal){
    if (goal == 0) return true;
    if (len == 0) return false;
    return
        naive_decision(arr + 1, len - 1, goal - arr[0])
        ||
        naive_decision(arr + 1, len - 1, goal);
}
```

La soustraction `goal - arr[0]` peut donner un nombre négatif, qui sera pris modulo 2^{64} puisqu'on utilise des `uint64_t`. Cependant, tout le calcul reste valable modulo 2^{64}.

2. En notant $T(n)$ la complexité en temps dans le pire cas pour un tableau de taille n, on a immédiatement $T(n) \leqslant 2T(n-1) + A$ avec A une constante. On en déduit $T(n)/2^n - T(n-1)/2^{n-1} \leqslant A/2^n$, puis en sommant $T(n)/2^n - T(0) \leqslant \sum_{k=1}^{n} A/2^k$. La somme de droite est bornée, d'où $T(n) = O(2^n)$. On peut remarquer que, dans le cas où la somme est inaccessible, la complexité est bien de l'ordre de 2^n : ce grand-O est optimal.

Correction de l'exercice L.2 page 1

1.

```c
bool naive_solution_aux(T arr[], int len, T goal, bool sol[]){
    if (goal == 0) return true;
    if (len == 0) return false;
    sol[0] = true;
    if (naive_solution_aux(arr + 1, len - 1, goal - arr[0], sol + 1)) {
        return true;
    } else {
        sol[0] = false;
        return naive_solution_aux(arr + 1, len - 1, goal, sol + 1);
    }
}
```

2. Pas de difficulté, le travail est fait par la fonction auxiliaire :

```c
bool naive_solution_aux(T arr[], int len, T goal, bool sol[]){
    if (goal == 0) return true;
    if (len == 0) return false;
    sol[0] = true;
    if (naive_solution_aux(arr + 1, len - 1, goal - arr[0], sol + 1)) {
        return true;
    } else {
        sol[0] = false;
        return naive_solution_aux(arr + 1, len - 1, goal, sol + 1);
    }
}
```

Correction de l'exercice L.3 page 2

```c
int main(int argc, char *argv[]){
    FILE *input_file = stdin;
    FILE *output_file = stdout;

    if (argc > 1) input_file = fopen(argv[1], "r");
    if (input_file == NULL) {
        fprintf(stderr, "File %s not found.\n", argv[1]);
        exit(EXIT_FAILURE);
    }

    if (argc > 2) output_file = fopen(argv[2], "w");
    if (output_file == NULL) {
        fprintf(stderr, "Cannot open %s for writing.\n", argv[2]);
    }

    int len;
    uint64_t goal;
    uint64_t *elements = read_elements(input_file, &len, &goal);

    bool *solution = naive_solution(elements, len, goal);
    if (solution) {
        fprintf(output_file, "Yes\n");
        print_solution(output_file, elements, len, solution);
    } else {
        fprintf(output_file, "No\n");
    }

    free(elements);
    free(solution);
    return EXIT_SUCCESS;
}
```

Correction de l'exercice L.4 page 3

```c
void compute_sums(T elements[], T set, int i, T partial, T sums[]){
    if (i < 0) {
        sums[set] = partial;
        return;
    }
    T with_i = set | (1ull << i);
    compute_sums(elements, with_i, i - 1, partial + elements[i], sums);
    compute_sums(elements, set, i - 1, partial, sums);
}
```

Détail que vous n'avez pas à connaître mais qu'il faut comprendre : pour produire un `uint64_t` avec le bit i à 1, il est nécessaire de faire `1ull << i` et pas `1 << i`. En effet, dans le deuxième cas, le début du calcul (avant le « ou » logique) se fait en `int`, ce qui est incorrect dès que $i \geqslant 31$. Le suffixe `ull` (`unsigned long long`) garantit qu'on travaille directement sur des entiers non signés 64 bits.

Correction de l'exercice L.5 page 3

1. On obtient exactement la même relation de récurrence que pour `naive_decision`, et donc une complexité temporelle en $O(2^i)$.
2. Supposons n pair pour éviter les parties entières. On a deux tableaux de taille $2^{n/2}$ contenant les sommes que l'on peut obtenir à partir de $n/2$ premiers (respectivement $n/2$ derniers) éléments de X. Chacun de ces tableaux a été obtenu en temps $O(2^{n/2})$, mais si l'on effectue toutes les sommes possibles entre un élément de $s(A)$ et un de $s(B)$, il y a $2^{n/2} \times 2^{n/2} = 2^n$ sommes à considérer. On obtient donc une complexité totale en $O(2^n)$, ce qui n'est pas mieux que la méthode naïve.
3. On exploite le caractère trié, comme vu lors d'un des premiers devoirs surveillés de l'année :

```c
bool exists_sum(T goal, T sA[], int n, T sB[], int p){
    T len_sA = 1ull << n;
    T len_sB = 1ull << p;
    size_t iA = 0;
    size_t iB = len_sB - 1;
    while (iA < len_sA && iB > iB - 1) {
        T s = sA[iA] + sB[iB];
        if (s == goal) return true;
        if (s < goal) iA++;
        else iB--;
    }
    return false;
}
```

Correction de l'exercice L.6 page 4

1. On commence par transtyper (*caster*, dans la langue courante) le pointeur a vers un `uint64_t*` (en conservant le qualificatif `const`), puis l'on déréférence ce pointeur pour obtenir un `uint_64`.
2. On travaille sur des entiers non signés 64 bits : si $x < y$, on obtiendrait un résultat très grand et positif (puisqu'on travaille modulo 2^{64}). Ce résultat serait ensuite convertit en `int` (type de retour de la fonction) de manière implicite. Sachant que le type `int` est signé, et trop petit pour représenter la valeur (en général), cette conversion peut donner un peu n'importe quoi. Si je lis bien le standard, c'est *implementation defined* et pas *undefined behavior*,

mais dans tous les cas il y a assez peu de chance que ça donne ce dont on a envie.

3.

Correction de l'exercice L.7 page 5

Correction de l'exercice L.8 page 5

Correction de l'exercice L.9 page 5