# Subset sum

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.

— 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

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.

   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).

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);
   ```

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 $X = x_0, \ldots, x_{n-1}$ en deux parties $A = x_0, \ldots, x_{\lfloor n/2 \rfloor - 1}$ et $B = x_{\lfloor n/2 \rfloor}, \ldots, x_{n-1}$ ;

— 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` $x$.

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.

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).

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 1;
  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);
```

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 ».*

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.

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