Cette *vingt-septième* colle vous fera travailler sur un problème d'optimisation combinatoire et sa résolution (approchée) par un algorithme glouton.

Le problème de décision de 2-partition

On s'intéresse au problème *de décision* **2-partition** suivant :

— **Entrée :** E un multi-ensemble de $n \geqslant 2$ entiers naturels, $E = \{e_1, \dots, e_n\}$ ($\forall i \in \{1, \dots, n\}$, $e_i \in \mathbb{N}$). Un multi-ensemble signifie juste qu'il peut y avoir des doublons dans E, autrement dit en informatique c'est une liste de nombres naturels.

— **Sortie :** existe-t-il un ensemble d'indices $I \subset \{1, \dots, n\}$ tel que $\sum_{i \in I} e_i = \sum_{i \in \{1,\dots,n\} \setminus I} e_i$.

D'un point de vue pratique, pour prendre un premier exemple ludique, ce problème peut par exemple modéliser un groupe de deux enfants (ex. Hansel et Gretel) qui reviennent de la chasse aux bonbons un soir d'Halloween : ils et elles ont récupérés n bonbons, chacun ayant un valeur calorique de e_i kcal connue, et souhaitent se les partager de telle sorte que Hansel reçoive les bonbons de l'ensemble d'indices I et Gretel ceux de l'ensemble d'indices complémentaire, $\{1, \dots, n\} \setminus I$, et que tous les deux reçoivent exactement la même somme totale de calories.

— Q1. Donner un exemple tout simple d'instance du problème qui admet une solution, et donner I.

— Q2. Faire de même avec un exemple d'instance qui n'admet pas de solution (avec quand même $n \geqslant 2$ nombres).

Avant d'étudier un éventuel algorithme efficace pour résoudre ce problème (ou une variante), on peut chercher à savoir si une résolution naïve peut suffire (même si elle sera peut-être très inefficace).

— Q3. Expliquer comment résoudre ce problème par exploration exhaustive (sans rentrer trop dans les détails).

— Q4. Sur une instance E de taille n quelconque, combien y a-t-il d'ensemble I de taille au moins 1 et au plus $n-1$? Notons ce nombre $g(n)$, et supposons avoir une méthode efficace pour les énumérer en temps $\mathcal{O}(g(n))$, quelle sera la complexité temporelle de l'algorithme donné en Q3. en fonction de n ?

Problème d'optimisation associée

On peut associer à ce problème de décision (**2-partition**) le problème *d'optimisation* (**2-partition-opt**) suivant :

— **Entrée :** E un multi-ensemble de $n \geqslant 2$ entiers naturels, $E = \{e_1, \dots, e_n\}$.

— **Sortie :** trouver un ensemble I qui minimise la quantité suivante : $f_E(I) = \max\big(\sum_{i \in I} e_i, \sum_{i \in \{1,\dots,n\} \setminus I} e_i\big)$ (sur l'inconnue $I \subset \{1, \dots, n\}$).

Ce problème peut être abordé comme un problème d'ordonnancement de n tâches indépendantes, de temps d'exécution donnés et connus à l'avance $e_1, \dots, e_n$, sur deux machines identiques. L'objectif est de minimiser (sur l'inconnue I des indices des tâches à affecter à la machine 1), le temps total d'exécution des tâches de l'ensemble E.

Pour comprendre en quoi ce problème est lié au précédent, voici une question à traiter :

— Q5. Supposons que l'on dispose d'un algorithme A' qui résolve le problème d'optimisation **2-partition-opt** et trouve toujours une solution optimale I^* sur n'importe quelle instance E, comment peut-on l'utiliser pour résoudre le problème de décision **2-partition** ?

Exemples à traiter à la main

On considère l'exemple suivant : $E = \{3, 1, 1, 2, 2, 1\}$.

— Q6. Trouver une solution au problème de décision **2-partition**.

  — Pouvez-vous en trouver une deuxième ? (si la première solution est I_1, répondre $I_2 = \{1, \dots, n\} \backslash I_1$ ne suffit pas !)

  — Que pouvez-vous déduire de la solution au problème d'optimisation **2-partition-opt** ?

On considère un autre exemple : $E' = E \cup \{1\} = \{3, 1, 1, 2, 2, 1, 1\}$.

— Q7. Cette instance E' admet-elle une solution au problème **2-partition** ?

  — Trouver une solution optimale au problème d'optimisation **2-partition-opt** pour cette instance E'.

Une condition nécessaire

On considère encore l'exemple suivant : $E = \{3, 1, 1, 2, 2, 1\}$.

— Q7. Calculer la somme totale $\sum_{i \in \{1,\dots,n\}} e_i$. Quelle est sa parité ?

— Q8. Dans le cas général, si l'instance E a une solution au problème **2-partition**, que peut-on dire sur la valeur la somme totale $\sum_{i \in \{1,\dots,n\}} e_i$?

— Q9. Cette condition nécessaire sur la somme totale est-elle suffisante ?

En fait, ce problème de décision de **2-partition** et son problème d'optimisation associée sont connus pour être NP-complets. Sans rentrer dans les détails (vous en aurez plus l'an prochain en MPI), il faut retenir que pour l'instant la communauté de la recherche en algorithmique considère qu'*un problème NP-complet est un problème difficile à résoudre exactement.* Le résultat $P =? NP$ est un d'ailleurs un des plus grands problèmes ouverts en informatique.

Algorithme glouton

On va désormais s'intéresser à une stratégie gloutonne qui aura l'avantage d'être simple à mettre en place, et très efficace en temps (en fonction du paramètre $n = |E|$).

- Trier éventuellement les tâches selon un certain ordre (par exemple par temps d'exécution e_i *décroissant*) ;
- On alloue la première tâche e_1 à la machine 1 (i.e. on ajoute l'indice 1 à l'ensemble I_1),
- Puis pour $i = 2$ à n, on s'intéresse à la tâche i de durée e_i :
  - On calcule la durée totale des tâches allouées aux deux machines 1 et 2 : $\sum_{i \in I_1} e_i$ et $\sum_{i \in I_2} e_i$,
  - On alloue la tâche i à la machine qui est actuellement la moins chargée ($\star$).
- On renvoie l'ensemble d'indice I_1 (et $I_2 = \{1, \dots, n\} \backslash I_1$ si besoin).

Quelle différence y a-t-il entre les deux stratégies ?

- La version sans tri peut s'appliquer sur des tâches qui arrivent dynamiquement dans le système (car pas besoin de les connaître toutes à l'avance). On dit que l'algorithme est en temps réel, ou *au fil de l'eau* (*online algorithm* en anglais).
- La version avec tri nécessite en revanche de connaître tous les temps d'exécution au préalable.

Des questions sur l'algorithme

- Q10. Si le critère de tri est assez simple pour être implémenté par un tri par comparaison sur les entiers e_i (par exemple par temps e_i décroissant), comment peut-on trier efficacement cette liste E de n entiers ?
  - Quel temps cela prendra en fonction de n, avec un algorithme efficace ?
  - Citer au moins deux algorithmes vus dans l'année qui ne sont pas parmi les plus efficaces.
  - Même question avec des algorithmes parmi les plus efficaces.
- Q11. Expliquer comment on peut éviter d'avoir à recalculer les deux sommes $\sum_{i \in I_1} e_i$ et $\sum_{i \in I_2} e_i$ en temps linéaire en $|I_1|$ et $|I_2$ à chaque étape ?
  - Avec cette optimisation, et en ignorant l'étape préliminaire de tri de E selon un certain critère, quelle sera la complexité temporelle totale de l'algorithme glouton ?
  - Est-ce que c'est efficace ? Pouvait-on espérer plus efficace ?
- Q12. Pour la ligne notée $\star$ ("On alloue la tâche i à la machine qui est actuellement la moins chargée"), il peut avoir une ambiguïté si les deux sommes sont égales. Quel choix simple proposez-vous pour enlever l'ambiguïté ?

Implémentation - langage libre OCaml / C

Pour conclure, vous allez implémenter cet algorithme, dans le langage de votre choix : en OCaml ou en C. Pour vous aider, voici les signatures que peuvent avoir la fonction `deux_partition_gloutonne` dans les deux langages.

— En OCaml : on renvoie un couple des valeurs $[e_i : i \in I_1]$ du premier ensemble et $[e_i : i \in I_1]$ du second ensemble.

```
deux_partition_gloutonne : int list -> (int list * int list)
```

— En C : on ne renvoie rien, mais à la fin on affiche à l'écran la liste des valeurs du premier ensemble, puis du second ensemble (pour simplifier).

```
void deux_partition_gloutonne(int n, int taches[n]);
```

— Q13. Expliquer comment vous représenter une instance du problème d'optimisation **2-partition-opt** dans le langage choisi.

— Q14. Définir au moins deux exemples d'instances du problème (utiliser ceux des questions Q1 et Q2, ou Q7 et Q8).

— Q15. Écrire une fonction `deux_partition_gloutonne`.

  — En OCaml, on peut implémenter une fonction `compare_tache : int -> int -> int` telle que `compare_tache ei ej` renvoie `0` en cas d'égalité, `-1` si `ei > ej` (on veut trier par temps d'exécution e_i décroissant), et `+1` sinon.
  Ensuite, le tri initial peut se faire avec un appel à `List.sort compare_tache liste_taches` (qui renvoie une nouvelle liste), ou `Array.sort compare_tache tableau_taches` (qui change le tableau en place).

  — En C, pour simplifier on peut supposer que les tâches sont déjà données dans l'ordre voulu.

— Q16. Tester votre fonction sur les exemples de la Q14.

