Exercice 3

On s'intéresse au problème d'optimisation `Rangement` : on dispose de n objets de poids respectifs $p_1, p_2, \ldots, p_n$ et on souhaite ranger ces objets dans des boîtes, sans qu'une seule boîte ne dépasse un poids maximal P (commun à toutes les boîtes), tout en minimisant le nombre de boîtes utilisées.

L'algorithme *First-fit* procède comme suit :

- on ouvre une boîte
- on considère les objets par indice croissant ;
- si on peut mettre l'objet suivant dans la boîte ouverte, on le fait, sinon on ferme la boîte et on ouvre une nouvelle boîte pour y mettre l'objet.

Pour créer un exécutable avec les fichiers fournis, on pourra utiliser la commande `gcc -o nom_exec abr.c rangement.c`. Seul le fichier `rangement.c` doit être modifié.

1. Écrire une fonction `int first_fit(int* poids, int n, int P)` qui renvoie le nombre de boîtes utilisées pour ranger les objets dont les poids sont donnés par le tableau `poids` de taille `n`, avec un poids maximal `P`, selon l'algorithme *First-fit*.
2. Montrer que la somme des poids de deux boîtes consécutives est strictement supérieure à P. En déduire que l'algorithme *First-fit* est une 2-approximation du problème `Rangement`.

Dans la version décisionnelle du problème, on se fixe un nombre de boîtes maximal m, et on se pose la question s'il existe un rangement des objets utilisant au plus m boîtes. On admet que le problème suivant est NP-complet, `Partition` :

* **Instance** : n valeurs entières positives $x_1, \ldots, x_n$.
* **Question** : existe-t-il un ensemble $I \subseteq [\![1, n]\!]$ tel que $\sum_{i \in I} x_i = \sum_{i \notin I} x_i$?

3. Montrer que la version décisionnelle de `Rangement` est NP-complète.

On se donne une structure d'arbres binaires de recherche par :

```
struct ABR {
    int valeur;
    struct ABR* gauche;
    struct ABR* droite;
};
typedef struct ABR abr;
```

telle que si a est un arbre implémenté par un objet `a` de type `abr*`, alors `a->valeur`, `a->gauche` et `a->droite` représente l'étiquette de la racine, l'enfant gauche et l'enfant droit de a respectivement.

Les fonctions fournies dans le fichier `abr.c` permettent d'insérer, de supprimer et de mettre à jour une valeur respectivement. On ne demande pas d'étudier ces fonctions et on admet qu'elles ont une complexité linéaire en la hauteur de l'arbre. On dispose également d'une fonction permettant de libérer la mémoire allouée par un arbre.

4. Écrire une fonction `int min_plus_grand(abr* a, int x)` qui prend en argument un arbre a et un entier x et renvoie la valeur minimale parmi les nœuds de a plus grands que x, ou `INT_MAX` s'il n'existe pas de tel nœud. Cette fonction devra avoir une complexité linéaire en la hauteur de l'arbre.

L'algorithme *Best-fit* suit le principe suivant :

- on ouvre une boîte ;
- on considère les objets par indice croissant ;
- si on peut mettre l'objet suivant dans une ou plusieurs boîtes, on choisit celle avec le plus petit poids restant, sinon on ouvre une nouvelle boîte pour y mettre l'objet.

5. Écrire une fonction `int best_fit(int* poids, int n, int P)` qui renvoie le nombre de boîtes utilisées pour ranger les objets selon l'algorithme *Best-fit*.
6. Quelle est la complexité dans le pire cas de la fonction `best_fit` ? Comment modifier la structure d'arbre binaire de recherche pour améliorer la complexité ? On ne demande pas d'implémenter cette solution.
7. Montrer que l'algorithme *Best-fit* est une $1,75$-approximation de `Rangement`.