Devoir surveillé d'informatique

Consignes

- Les programmes demandés doivent être écrits en C ou en OCaml suivant l'exercice. Dans le cas du C, on suppose que les librairies standards usuelles (`<stdio.h>`, `<stdlib.h>`, `<stdbool.h>`) sont déjà importées.
- On pourra toujours librement utiliser une fonction demandée à une question précédente même si cette question n'a pas été traitée.
- Veillez à présenter vos idées et vos réponses partielles même si vous ne trouvez pas la solution complète à une question.
- La clarté et la lisibilité de la rédaction et des programmes sont des éléments de notation.

❑ Exercice 1 : *Base de données et* SQL

CAPES NSI 2021, épreuve 1

On s'intéresse dans cette partie à un site Internet d'échange de supports de cours entre enseignants de MP2I/MPI. Chaque personne désirant proposer ou récupérer du contenu doit commencer par se créer un compte sur ce site et peut ensuite accéder à du contenu ou en proposer.

Ce site repose sur une base de données contenant en particulier une table, nommée `ressources`. Elle possède un enregistrement par document téléversé sur le site. Ses attributs sont :

— `id`, un identifiant numérique, unique pour chaque ressource ;

— `owner`, le pseudo de la personne ayant créé la ressource ;

— `annee`, l'année de publication de la ressource ;

— `titre`, une chaine de caractères décrivant la ressource ;

— `type`, chaine de caractères pouvant être cours, ds, tp ou td.

Voici un extrait de cette table :

id	owner	annee	titre	type
4	dknuth	2020	Machine à décalage	cours
13	alovelace	2022	Intelligence artificielle	td
...	...	...	...	...

Q1– Écrire une requête SQL permettant de connaitre tous les titres des ressources déposées par « `jclarke` » classées par année de publication croissante.

```
SELECT titre
FROM ressources
WHERE owner="jclarke"
ORDER BY annee ASC;
```

Q2– Écrire une requête SQL permettant de connaitre le nombre total de ressources de type cours présentes sur le site.

```
SELECT COUNT(*)
FROM ressources
WHERE type = "cours";
```

Q3– Que fait la requête suivante : ?

```
SELECT R.owner
FROM Ressources AS R
WHERE R.type = 'td'
GROUP BY R.owner
ORDER BY COUNT(*) DESC
LIMIT 3
```

> Cette requête permet d'afficher les trois premiers propriétaires de ressources ayant posté le plus de ressources de type `td`

Cette base de données contient également une table `utilisateurs` qui contient les informations sur les utilisateurs du site. Elle possède un enregistrement par utilisateur. Ses attributs sont :

— `nom`, le nom de l'utilisateur (clé primaire) ;

— `mdp`, le mot de passe de l'utilisateur.

— `email`, l'adresse email de l'utilisateur.

— `naissance`, l'année de naissance de l'utilisateur.

Voici un extrait de cette table :

nom	mdp	email	naissance
dknuth	chepas123	dknuth@bigboss.com	1938
...	...	...	...

L'attribut `owner` de la table `ressources` est une clé étrangère qui référence l'attribut `nom` de la table `utilisateurs`.

Q4– Ecrire une requête SQL permettant de lister toutes les ressources déposées par des utilisateurs nés après 2000.

```
SELECT * FROM ressources
JOIN utilisateurs
ON ressources.owner = utilisateurs.nom
WHERE utilisateurs.naissance>2000
```

Q5– Ecrire une requête SQL permettant de lister tous les utilisateurs n'ayant déposé aucune ressource.

```
SELECT nom FROM utilisateurs
EXCEPT
SELECT owner FROM ressources
```

❑ Exercice 2 : *Logique et calcul des propositions*

🎓 CCP 2015

De nombreux travaux sont réalisés en Intelligence Artificielle pour construire un programme qui imite le raisonnement humain et soit capable de réussir le test de Turing, c'est-à-dire qu'il ne puisse pas être distingué d'un être humain dans une conversation à l'aveugle. Vous êtes chargé(e)s de vérifier la correction des réponses données par un tel programme lors des tests de bon fonctionnement. Dans le scénario de test considéré, le comportement attendu est le respect de la règle suivante : pour chaque question, le programme répondra par trois affirmations et parmi ces affirmations une seule sera correcte (et donc les deux autres fausses).
Nous noterons A_1 , A_2 et A_2 les propositions associées aux affirmations effectuées par le programme.

Q6– Représenter le comportement attendu sous la forme d'une formule du calcul des propositions qui dépend de A_1, A_2 et A_3.

> Le comportement attendu est que parmi les trois affirmations, une seule est vraie. On peut donc écrire la formule suivante :
> $(A_1 \wedge \neg A_2 \wedge \neg A_3) \vee (\neg A_1 \wedge A_2 \wedge \neg A_3) \vee (\neg A_1 \wedge \neg A_2 \wedge A_3)$.

■ Partie I : Premier cas

Vous demandez au programme : « *Quels éléments doivent contenir les aliments que je dois consommer pour préserver ma santé ?* »
Il répond les affirmations suivantes :

A_1 : Consommez au moins des aliments qui contiennent des glucides mais pas de lipides !

A_2 : Si vous consommez des aliments qui contiennent des glucides, alors ne consommez pas d'aliments qui contiennent des lipides !

A_3 : Ne consommez aucun aliment qui contient des lipides

Nous noterons G, respectivement L, les variables propositionnelles qui correspondent au fait de consommer des aliments qui contiennent des glucides, respectivement des lipides.

Q7– Exprimer A_1, A_2 et A_3 sous la forme de formules du calul des propositions. Ces formules peuvent dépendre des variables G et L.

- $A_1 : G \wedge \neg L$
- $A_2 : G \rightarrow \neg L$
- $A_3 : \neg L$

Q8– En utilisant le calcul des propositions (résolution avec les formules de De Morgan), déterminer ce que doivent contenir les aliments que vous devrez consommer pour préserver votre santé.

Comme

- $\neg A_1 = \neg G \vee L$
- $A_2 \equiv \neg G \vee \neg L$ et donc $\neg A_2 = G \wedge L$
- $\neg A_3 = L$

La règle énoncée conduit à :
$(G \wedge \neg L \wedge G \wedge L \wedge L) \vee ((\neg G \vee L) \wedge (\neg G \vee \neg L) \wedge L) \vee ((\neg G \vee L) \wedge G \wedge L \wedge \neg L)$
La première et la troisième parenthèsent valant *Faux*, cette formule devient donc :
$((\neg G \vee L) \wedge (\neg G \wedge L))$
$\neg G \wedge L$
Donc on doit consommer des lipides et pas de glucides.

■ **Partie II** : Deuxième cas

Vous demandez au programme : « *Quelles activités dois-je pratiquer si je veux préserver ma santé ?* »
Suite à une coupure de courant, la dernière information est interrompue.

A_1 : Si vous faites des activités sportives alors prenez du repos !

A_2 : Si vous ne faites pas d'activités intellectuelles, alors ne prenez pas de repos !

A_3 : Prenez du repos ou faites des activités ... !

Nous noterons S, I et R les variables propositionnelles qui correspondent au fait de faire des activités sportives, des activités intellectuelles, et de prendre du repos.

Q9– Exprimer A_1, A_2 et A_3 sous la forme de formules du calul des propositions. Ces formules peuvent dépendre des variables S, I et R. Pour A_3, on indiquera des pointillés pour signifier que la réponse a été interrompue.

- $A_1 : S \rightarrow R$ c'est à dire $A_1 = \neg S \vee R$
- $A_2 : \neg I \rightarrow \neg R$ c'est à dire $A_2 = I \vee \neg R$
- $A_3 : R \vee ...$

Q10– En utilisant le calcul des propositions (résolution avec les tables de vérité), déterminer quelle(s) activités vous devez pratiquer pour préserver votre santé. L'information A_3 étant incomplète ou pourra envisager les deux cas suivants : elle se termine par "sportives" ou "intellectuelles" et réaliser la table de vérité dans chacun des cas.

S	I	R	A_1	A_2	A_3 si $\cdots = S$	Règle si $\cdots = S$	A_3 si $\cdots = I$	Règle si $\cdots = I$
0	0	0	1	1	0	0	0	0
0	0	1	1	0	1	0	1	0
0	1	0	1	1	0	0	1	0
0	1	1	1	1	1	0	1	0
1	0	0	0	1	1	0	0	1
1	0	1	1	0	1	0	1	0
1	1	0	0	1	1	0	1	0
1	1	1	1	1	1	0	1	0

On constate donc que si A_3 est $R \vee S$ alors la règle qui veut qu'une seule des 3 affirmations soit vraie n'est pas respectée.
Par contre si A_3 est $R \vee I$ alors la règle est vérifiée avec S vraie et I et R fausses. Donc, on doit faire des activités sportives et ne pas prendre du repos ni faire d'activités intellectuelles.

❑ Exercice 3 : *Implémentation d'un tableau associatif*

🎓 CCMP 2025, MPI

Les fonctions demandées dans cet exercice doivent être écrites en langage C.

Définitions : Soit A un arbre binaire de recherche, on notera avec une lettre minuscule (par exemple x ou y) un noeud de A et avec une lettre majuscule (par exemple T_1, T_2 ou T_3) un sous arbre de A. Soit x un noeud, on notera x_g et x_d respectivement le fils gauche et le fils droit de x. La hauteur d'un arbre x sera noté $h(x)$ et pour arbre de racine x, on définit la valeur d'équilibre de x noté $e(x)$ comme la différence entre la hauteur du sous arbre gauche et celle du sous arbre droit, c'est à dire :

$$e(x) = h(x_g) - h(x_d)$$

On définit un noeud d'un arbre binaire de recherche par la structure suivante :

```c
struct abrNoeud_s
{
    char *cle;
    int valeur;
    struct abrNoeud_s *fils_gauche;
    struct abrNoeud_s *fils_droit;
};
typedef struct abrNoeud_s abrNoeud;
```

On utilisera la valeur du champ `clef` pour ordonner les noeuds de l'arbre en utilisant l'ordre alphabétique.

Q11– Nous nous intéressons aux valeurs du champ `clef` de la structure ci-dessus. Expliquer comment une chaine de caractères est représentée en mémoire en C. Combien d'octets sont nécessaires pour représenter la chaine *arbre* par exemple ?

En C, une chaine de caractères est représentée par un tableau de caractères terminé par le caractère nul `'\0'`. Par exemple la chaine *arbre* est représentée par le tableau de 6 caractères : `'a', 'r', 'b', 'r', 'e', '\0'`. Il faut donc 6 octets pour représenter cette chaine.

Q12– Sachant que le code ASCII du caractère `'a'` est 97 quelle est sa représentation en binaire ? Quelle est sa représentation en hexadécimal ?

$\overline{97}^{10} = \overline{1100001}^{2} = \overline{61}^{16}$

Q13– Sans utiliser les fonctions de la librairie `string.h`, écrire une fonction en C, `copie_chaine` qui renvoie une copie de la chaine de caractères passée en paramète. La signature de la fonction est :
`char* copie_chaine(char* s)`.
On supposera que la chaine de caractères passée en paramètre n'est jamais NULL.

```c
char *copie_chaine(char *s)
{
    // On calcule la longueur l de la chaine
    int l = 0;
    while (s[l] != '\0')
    {
        l++;
    }
    // On alloue l'espace pour la nouvelle chaine
    char *s2 = malloc((l + 1) * sizeof(char));
    // On recopie caractère par caractère
    for (int i = 0; i <= l; i++)
    {
        s2[i] = s[i];
    }
    return s2;
}
```

Q14– Ecrire une fonction en C, `abr_creer_noeud` qui crée un noeud à partir d'une clef, d'une valeur et de deux sous arbres et qui renvoie ce nouveau noeud. La signature de la fonction est :
`abrNoeud* abr_creer_noeud(char* clef, int valeur, abrNoeud* fg, abrNoeud* fd)`.
Le champ `clef` du noeud crée doit contenir une *copie* de la chaine de caractères passée en paramètre. Par contre, il n'est pas nécessaire de dupliquer les sous-arbres passés en paramète.

```c
abrNoeud *creer_noeud(char *cle, int valeur, abrNoeud *fg, abrNoeud *fd)
{
    abrNoeud *noeud = malloc(sizeof(abrNoeud));
    noeud->cle = copie_chaine(cle);
    noeud->valeur = valeur;
    noeud->fils_gauche = fg;
    noeud->fils_droit = fd;
    return noeud;
}
```

Q15– Ecrire une fonction, `hauteur` qui renvoie la hauteur de l'arbre binaire de recherche passé en paramètre. La signature de la fonction est : `int hauteur(abrNoeud* a)`.

```c
int hauteur(abrNoeud *ab)
{
    if (ab == NULL)
    {
        return -1;
    }
    int hg = hauteur(ab->fils_gauche);
    int hd = hauteur(ab->fils_droit);
    if (hg > hd)
    {
        return 1 + hg;
    }
    else
    {
        return 1 + hd;
    }
}
```

Q16– Ecrire une fonction, `equilibre` qui renvoie la valeur d'équilibre de l'arbre binaire de recherche passé en paramètre. La signature de la fonction est : `int equilibre(abrNoeud* a)`.

```c
int equilibre(abrNoeud *ab)
{
    int h_g = hauteur(ab->fils_gauche);
    int h_d = hauteur(ab->fils_droit);
    return (h_g - h_d);
}
```

Q17– Pour un arbre de taille N, quelle est la complexité de la fonction `equilibre` ? Sans proposer de code, expliquer quelles seraient les modifications à apporter à la structure de données et/ou à l'algorithme pour améliorer cette complexité

La complexité de la fonction `hauteur` est linéaire en fonction de la taille de l'arbre, car chaque sommet de l'arbre est parcouru une fois, donc la complexité de la fonction `equilibre` est un $\mathcal{O}(N)$. Pour améliorer cette complexité, on pourrait stocker la hauteur de chaque noeud dans la structure de données. Ainsi, il suffirait de renvoyer la différence entre les hauteurs des sous-arbres gauche et droit du noeud passé en paramètre et cela pourrait être fait en $\mathcal{O}(1)$.

Q18– Ecrire une fonction récursive `abr_rechercher` qui, étant donné un arbre binaire de recherche dont le noeud racine est passé en paramètre ainsi qu'une clef, renvoie la valeur associée à cette dernière. On considérera que la clef donnée est toujours dans l'arbre binaire de recherche et on rappelle qu'en C l'opérateur `==` permet de comparer des entiers ou des pointeurs mais pas de comparer le contenu de deux chaines de caractères. La signature de la fonction est : `int abr_rechercher(abrNoeud* a, char* clef)`.

```c
int compare_chaine(char *s1, char *s2)
{
    int i = 0;
    while (s1[i] != '\0' && s2[i] != '\0')
    {
        if (s1[i] < s2[i])
        {
            return -1;
        }
        else if (s1[i] > s2[i])
        {
            return 1;
        }
        i++;
    }
    if (s1[i] == '\0' && s2[i] == '\0')
    {
        return 0;
    }
    else if (s1[i] == '\0')
    {
        return -1;
    }
    else
    {
        return 1;
    }
}

int abr_rechercher(abrNoeud *a, char *clef)
{
    // Ce cas ne devrait jamais se produire car on suppose la clé présente
    if (a == NULL)
    {
        return -1;
    }
    int cmp = compare_chaine(clef, a->cle);
    if (cmp == 0)
    {
        return a->valeur;
    }
    else if (cmp < 0)
    {
        return abr_rechercher(a->fils_gauche, clef);
    }
    else
    {
        return abr_rechercher(a->fils_droit, clef);
    }
}
```

Q19– Une autre implémentation possible d'un tableau associatif utilise une table de hachage. Expliquer succinctement le principe de cette implémentation.

On crée un tableau de taille M et on choisit une fonction de hachage à valeur dans $[\![0; M-1]\!]$. On associe à chaque clef k la case d'indice $h(k)$ du tableau. Si deux clefs k_1 et k_2 sont différentes, il est possible que $h(k_1) = h(k_2)$. Dans ce cas, on dit que les clefs k_1 et k_2 entrent en collision.

Q20– Ecrire une fonction `hachage` qui prend en argument une chaine de caractères et renvoie la somme des codes ASCII de cette chaine pondérée par la position de chaque caractère. C'est à dire que le code ASCII du premier caractère est multiplié par 1, le code du second par deux et ainsi de suite. Par exemple le hachage de la chaine `"abc"` renvoie $1 \times 97 + 2 \times 98 + 3 \times 99 = 590$ (on rapelle que le code ASCII de `'a'` est 97, celui de `'b'` 98, ...). La signature de la fonction est `int hachage(char *s)`.

```c
int hachage(char *s)
{
    int h = 0;
    int i = 0;
    while (s[i] != '\0')
    {
        h += (i + 1) * (int)s[i];
        i += 1;
    }
    return h;
}
```

Q21– Expliquer rapidement pourquoi il serait plus pertinent d'utiliser un type entier non signé pour la valeur renvoyée par la fonction `hachage`.

Comme un dépassement de capacité sur un type entier signé est un comportement indéfini (qui peut donner un résultat négatif), il est préférable d'utiliser un type entier non signé sur lequel les dépassements de capcacité sont bien définis (et donne le résultat modulo le plus grand entier représentable par le type).

Q22– Donner pour la fonction de hachage précédente, deux chaines de caractères de longueur 2 et écrites uniquement avec des minusucules qui entrent en collision.

On peut par exemple prendre les chaines `"ab"` et `"ba"`. En effet, on a :
$h(\text{"}ab\text{"}) = 1 \times 97 + 2 \times 98 = 293$ et $h(\text{"}ca\text{"}) = 1 \times 99 + 2 \times 97 = 293$. De façon générale, tous les couples d'entiers (a, b) et (c, d) de $[\![97; 122]\!]^2$ tels que $a + 2b = c + 2d$ conviennent, cette équation donne $c - a = 2(b - d)$.

❑ Exercice 4 : *Problème du sac à dos*

Les fonctions demandées dans cet exercice doivent être écrites en OCaml et on *s'interdit* dans cet exercice l'utilisation des aspects impératifs du langage (boucles et références notamment).
On dispose d'un sac à dos pouvant contenir un poids maximal noté P et de n objets ayant chacun un poids $(p_i)_{0 \leqslant i \leqslant n-1}$ et une valeur $(v_i)_{0 \leqslant i \leqslant n-1}$. On cherche à remplir le sac à dos de manière à maximiser la valeur totale des objets contenus dans le sac sans dépasser le poids maximal P. Par exemple, si on dispose des objets suivants :

- un objet de poids $p_0 = 4$ et de valeur $v_0 = 20$,
- un objet de poids $p_1 = 5$ et de valeur $v_1 = 28$,
- un objet de poids $p_2 = 6$ et de valeur $v_2 = 36$,
- un objet de poids $p_3 = 7$ et de valeur $v_3 = 50$,

et qu'on suppose que le poids maximal du sac est 10 alors un choix possible serait de prendre l'objet 3, aucun autre objet ne rentre alors dans le sac et la valeur du sac est de 50 avec un poids de 7. Une autre possibilité plus intéressante serait de choisir les objets 0 et 2, la valeur totale serait alors de 56 et le poids du sac de 10.

Dans toute la suite de l'exercice on supposera que les poids et les valeurs des objets ainsi que le poids maximal du sac sont des entiers (type `int`), et que donc un objet être représenté par un couple (poids, valeurs) `int*int` et une liste d'objets par le type `(int*int) list` de OCaml. Et on définit le type suivant afin de représenter un problème du sac à dos :

```
type pbsac =
  {
    pmax : int;
    objets : (int*int) list
  }
```

Ainsi, le problème donné en exemple ci-dessus correspond à

```
  let ex =
    {
      pmax = 10;
      objets = [(7, 35); (4, 18); (5, 20);]
    }
```

On propose de représenter un choix d'objets par une liste de booléens. Si le i-ème élément de la liste vaut `true` alors l'objet i est choisi, sinon l'objet i n'est pas choisi. Par exemple, pour les objets précédents, le choix de prendre uniquement l'objet 3 serait représenté par la liste `[false; false; false; true]` et le choix de prendre les objets 0 et 2 serait représenté par la liste `[true; false; true; false]`.

■ Partie I : Approche par recherche exhaustive

La recherche exhaustive consiste à énumérer tous les choix possibles d'objet et à calculer la valeur ainsi que le poids pour chaque choix. Parmi toutes les possibilités, on retient alors le choix qui maximise la valeur du sac sans dépasser le poids maximal.

Q23– Justifier rapidement que le nombre possible de choix d'objet est 2^n.

> Pour chaque objet, on a deux choix possibles, le prendre ou non, comme il y a n objets, le nombre total de choix est 2^n.

Q24– Ecrire une fonction `poids_valeur : (int*int) list -> bool list -> int*int` qui prend en arguments, une liste d'objets ainsi qu'un choix d'objet (sous la forme indiquée ci-dessus) et qui renvoie le poids et la valeur du sac correspondant à ce choix. Par exemple avec le problème du sac à dos `ex` définie ci-dessus, `poids_valeur(ex.objets, [true; false; true; false])` doit renvoyer `(10, 56)`. On supposera que la liste d'objets et le choix d'objet sont de même taille.

```
let rec poids_valeur objets choix =
  match objets, choix with
  | [], [] -> (0, 0)
  | [], _ -> failwith "Erreur : choix trop long"
  | _, [] -> failwith "Erreur : choix trop court"
  | (v,p)::reste, pris::rchoix -> let vr, pr = poids_valeur reste rchoix in
    if pris then v+vr, p+pr else vr, pr;;
```

Q25– Sans le programmer, donner un algorithme permettant de générer tous les choix possibles d'objets.

> On peut par exemple parcourir les entiers de 0 à 2^{n-1} et pour chacun d'eux construire la liste de booléens correspondante. Une autre possibilité est d'utiliser un compteur binaire initialisé à `false` sur tous les bits puis d'incrémenter ce compteur jusqu'à atteindre la valeur $2^n - 1$. Pour chaque valeur du compteur, on construit la liste de booléens correspondante.

Q26– Donner la complexité d'une méthode qui pour chaque choix possible d'objet calculerait la valeur du sac ainsi que son poids et renverrait le choix optimal.

> Il y a 2^n choix d'objets possibles, et pour chacun de ces choix on doit effectuer n opérations afin de calculer la valeur et le poids du sac. La complexité de cette méthode est donc en $\mathcal{O}(n2^n)$.

■ **Partie II** : Stratégie gloutonne

On considère la stratégie gloutonne suivante : on trie les objets par ordre décroissant de leur rapport valeur/poids et on les prend dans cet ordre jusqu'à ce que le poids maximal soit atteint.

Q27– Montrer à l'aide d'un contre exemple que cette stratégie ne permet pas toujours d'obtenir la valeur maximale.

> On peut utiliser la liste d'objets suivantes (déjà triée par rapport valeur/poids croissante :)
> — $(7\,;\,35)$ (rapport $= 5$)
> — $(4\,;\,18)$ (rapport $= 4.5$)
> — $(5\,;\,20)$ (rapport $= 4$)
>
> Si on suppose que le poids maximal du sac est 10, alors l'algorithme glouton prendra seulement l'objet 0 (pour une valeur de sac de 35) alors que la valeur maximale est de 38 en prenant les objets 1 et 2.

Q28– La documentation de la fonction `List.sort : ('a -> 'a -> int) -> 'a list -> 'a list` de la librairie standard d'OCaml indique que cette fonction trie une liste d'éléments en utilisant l'ordre défini par la fonction de comparaison passée en paramètre. Cette dernière renvoie 0 lorsque les deux éléments sont égaux, un entier positif lorsque le premier élément est plus grand et un entier négatif sinon. Cette documentation indique aussi que l'algorithme utilisé est le *tri fusion*. En déduire une fonction `tri : (int*int) list -> (int*int) list` qui trie une liste d'objets par ordre décroissant de leur rapport valeur/poids.

```
let rec tri objets =
  List.sort (fun (p1, v1) (p2, v2) -> if
    (float_of_int v1 /. float_of_int p1) > (float_of_int v2 /. float_of_int p2)
  then -1 else 1) objets;;
```

Q29– Ecrire une fonction `glouton : pbsac -> int`, qui prend en arguments un problème du sac à dos et qui renvoie la valeur maximale que l'on peut obtenir en appliquant la stratégie gloutonne.

```
let glouton pbsac =
  let objets_tries = tri pbsac.objets in
  let rec aux objets pmax =
    match objets with
    | [] -> 0
    | (p,v)::reste -> if p<=pmax then
        let vr = aux reste (pmax - p) in
        v + vr
      else aux reste pmax
  in
  aux objets_tries pbsac.pmax;;
```

Q30– Donner la complexité de la foncton `glouton` en fonction du nombre d'objets n.

> Comme la fonction de tri utilisée est le tri fusion, sa complexité est en $\mathcal{O}(n \log n)$. Ensuite, la fonction `glouton` parcourt la liste triée une fois pour calculer la valeur du sac, donc sa complexité est en $\mathcal{O}(n)$. La complexité de la fonction `glouton` est donc en $\mathcal{O}(n \log n)$.

■ **Partie III** : Approche par programmation dynamique

On propose de résoudre le problème du sac à dos par programmation dynamique. Pour tout $i \in [\![0; n]\!]$, on note $V(i, p)$ la valeur maximale que l'on peut obtenir avec les objets à partir de celui d'indice i et un poids maximal p. Par exemple s'il y a 5 objets (numérotés de 0 à 4), $v(2, 10)$ est le poids maximal atteint pour un sac de poids maximal 10 en ne considérant que les objets 2, 3 et 4.

Q31– Donner $V(i, 0)$ pour $i \in [\![0; n-1]\!]$ et $V(n, p)$ pour $p \in [\![0; P]\!]$.

> $V(i, 0)$ est la valeur maximal d'un sac de poids maximal nul c'est donc 0. $V(n, p)$ vaut 0 aussi car on n'utilise aucun objet (les numéros des objets vont de 0 à $n-1$)

Q32– Donner les relations de récurrence liant $v(i, p)$ à $v(i+1, p)$ et $v(i+1, p-p_i)$.
🛟 Indication : on pourra considérer deux cas, celui où l'objet i est n'est pas pris et celui où il l'est (dans ce cas on a nécessairement $p \geqslant p_i$).

> Pour déterminer $v(i, p)$, si $p \geqslant p_i$, on choisit la valeur maximale entre les deux possiblités suivantes :
> — ne pas prendre l'objet i, la valeur du sac est alors $v(i+1, p)$,
> — prendre l'objet i, la valeur du sac est alors $v_i + v(i+1, p-p_i)$.
>
> Sinon, on ne peut pas prendre l'objet i et donc la valeur du sac est $v(i, p) = v(i+1, p)$. On a donc la relation de récurrence suivante : $v(i, p) = \max(v(i+1, p), v_i + v(i+1, p-p_i))$ si $p \geqslant pi$ et $v(i, p) = v(i+1, p)$ sinon.

Q33– Écrire une fonction récursive `dynamique : pbsac -> int` qui prend en arguments un problème du sac à dos et qui renvoie la valeur maximale que l'on peut obtenir en appliquant la stratégie de programmation dynamique.

```
let dynamique pbsac =
  let rec aux objets pmax =
    match objets with
    | [] -> 0
    | (p,v)::reste ->
      if p<=pmax then
        let avec = v + aux reste (pmax-p) in
        let sans = aux reste pmax in
        max avec sans
      else aux reste pmax
  in
  aux pbsac.objets pbsac.pmax;;
```

Q34– Sans la programmer, expliquer une stratégie permettant de mémoïser les résultats déjà calculées de la fonction `dynamique` afin d'éviter de les recalculer.

> On pourrait enregistrer dans une table de hachage les résultats déjà calculés. Avec les notations de l'énoncé pour chaque valeur de $i \in [\![0; n-1]\!]$ et chaque valeur p de poids maximal autorisé, on enregistre dans la table de hachage la valeur de $V(i, p)$ afin d'y avoir accès en temps constant sans avoir à relancer la récursion.

❑ **Exercice 5** : *Tableaux autoréférents* 🎓 ORAUX CCINP 2024, MPI

Les fonctions demandées dans cet exercice doivent être écrites en langage C.
On dit qu'un tableau `tab` de taille n est *autoréférent* si pour tout entier `i` $\in [\![0; n-1]\!]$, `tab[i]` est le nombre d'occurences de `i` dans `tab`. Par exemple, le tableau `ex={ 1, 2, 1, 0 }` est autoréférent, en effet :

— `ex[0] = 1` et `0` apparaît bien une fois dans le tableau
— `ex[1] = 2` et `1` apparaît bien deux fois dans le tableau

— `ex[2] = 1` et 2 apparaît bien une fois dans le tableau

— `ex[3] = 0` et 3 n'apparaît pas dans le tableau

L'exercice traite de la recherche par retour sur trace d'un tableau autoréférent de taille donnée.

Q35– Justifier rapidement que dans un tableau autoréférent de taille n, chaque valeur doit être comprise entre 0 et n et que la somme des éléments du tableau doit être égale à n.

Si on considère un tableau autoréférent de taille n, alors on a $t[i]$ = nombre d'occurrences de i dans t, comme une valeur peut apparaître au maximum n fois, on a $t[i] \leq n$. De plus, t de taille n, il y a en tout n occurrences de valeurs dans t, c'est à dire que la somme des $t[i]$ vaut n.

Q36– Montrer que pour $n \in [\![1;3]\!]$, il n'existe aucun tableau auto référent de taille n.

Il suffit de tester les possibilités, en utilisant la question précédente.

— taille 1 : le seul tableau possible est $\{1\}$ et il n'est pas autoréférent.

— taille 2 : Comme $t[0]$ ne peut pas valoir 0, le seul tableau possible est $\{1,1\}$ et il n'est pas autoréférent.

— taille 3 : $t[0]$ ne peut pas valoir 0. Si $t[0] = 1$, alors soit $t[1] = 0$ et donc $t[2] = 2$ soit $t[2] = 0$ et $t[1] = 2$ or aucun de ces tableaux n'est autoréférent. Si $t[0] = 2$ alors $t[1] = 0$ et $t[2] = 0$ et ce ne tableau n'est pas autoréférent.

Q37– Déterminer un autre tableau autoréférent de taille 4 que celui donné en exemple.

— si $t[0] = 1$ alors 0 apparait et une seule fois, 1 apparait au moins une fois et la somme faisant 4, les possibilités sont $\{1,1,0,2\}$, $\{1,1,2,0\}$, $\{1,2,0,1\}$ et $\{1,2,1,0\}$. Seul le dernier tableau est autoréférent et c'est celui donné dans l'énoncé.

— si $t[0] = 2$ alors 0 apparait deux fois, donc les possibilités sont $\{2,2,0,0\}$, $\{2,0,2,0\}$ et $\{2,0,0,2\}$. Seul le tableau $\{2,0,2,0\}$ est autoréférent.

— si $t[0] = 3$ alors 0 apparait trois fois, impossible.

En conclusion, il n'y a qu'une seule autre possibilité que celle donnée dans l'enoncé, c'est le tableau $\{2,0,2,0\}$.

Q38– Soit `n`≥ 7, on définit le tableau `tab` de taille `n` par :

— `tab.(0) = n-4`

— `tab.(1) = 2`

— `tab.(2) = 1`

— `tab.(n-4) = 1`

— `tab.(i) = 0` si `i` $\notin \{0,1,2,n-4\}$

Prouver que `tab` est autoréférent

On vérifie :

— `tab.(0) = n-4` et toutes les cases du tableau valent `0` sauf 4 cases, celles d'indices 0 (car $n-4 \neq 0$), 1, 2 et $n-4$.

— `tab.(1) = 2` et `1` apparaît bien deux fois dans le tableau aux indices 2 et $n-4$. (car $n-4 \neq 1$)

— `tab.(2) = 1` et `2` apparaît bien une fois dans le tableau (à l'indice 1)

— `tab.(n-4) = 1` et `n-4` apparaît bien une seule fois dans le tableau car comme $n \leqslant 7$, $n-4 \leqslant 3$.

— `tab.(i) = 0` si `i` $\notin \{0,1,2,n-4\}$ et aucune de ces valeurs n'apparaît dans le tableau

Donc ce tableau est bien autoréférent.

Q39– Ecrire une fonction `est_autoreferent` qui prend en argument un tableau d'entiers (et sa taille) et renvoie `true` si ce tableau est autoréférent et `false` sinon. On attend une complexité en $\mathcal{O}(n)$ où n est la taille du tableau c'est à dire qu'on veut parcourir une seule fois le tableau.

```
1  bool est_autoreferent(int tab[], int n)
2  {
3      // On initialise les occurrences de toutes les valeurs à 0
4      int occ[n];
5      for (int k = 0; k < n; k++)
6      {
7          occ[k] = 0;
8      }
9      // On parcourt en incrémentant les occurrences
10     for (int i = 0; i < n; i++)
11     {
12         occ[tab[i]]++;
13     }
14     // On teste si le tableau est autoréférent
15     for (int i = 0; i < n; i++)
16     {
17         if (occ[i] != tab[i])
18         {
19             return false;
20         }
21     }
22     return true;
23 }
```

On cherche maintenant à construire un tableau autoréférent de taille `n` en utilisant un algorithme de recherche par retour sur trace (*backtracking*) qui valide une solution partielle construite jusqu'à un index `i` donné, on propose pour cela la fonction récursive suivante qui utilise la fonction de validation partielle `valide` qui sera écrite plus loin et qui essaye d'affecter une valeur valide à l'indice `i` puis de poursuivre la construction du tableau :

```
1  bool autoreferent(int tab[], int n, int i)
2  {
3      if (i == n)
4      {
5          return est_autoreferent(tab,n);
6      }
7      for (int k = .......)
8      {
9          tab[i] = k;
10         if (valide(tab, n, i))
11         {
12             if (autoreferent(........))
13             {
14                 return .....;
15             }
16         }
17     }
18     return ......;
19 }
```

Q40– Compléter les lignes 7, 12, 14, et 18 de la fonction précédente.

```
bool autoreferent(int tab[], int n, int i)
{
    if (i == n)
    {
        return est_autoreferent(tab,n);
    }
    for (int k = 0; k <= n; k++)
    {
        tab[i] = k;
        if (valide(tab, n, i))
        {
            if (autoreferent(tab, n, i + 1))
            {
                return true;
            }
        }
    }
    return false;
}
```

Q41– On sait que la somme des éléments d'un tableau autoréférent de taille `n` est `n`. D'autre part, si après avoir affecté la case d'indice `i`, il y déjà strictement plus d'occurrences d'une valeur `k` comprise entre 0 et `i` que la valeur de `t[k]` alors la solution partielle n'est pas valide. En déduire une fonction `valide` de signature `bool valide(int tab[], int n, int i)` qui prend en argument un tableau d'entiers, sa taille et un indice `i` et qui renvoie `true` si la ce tableau peut-être complété en un tableau autoréférent.

```c
bool valide(int tab[], int n, int i)
{
    int s = 0;
    for (int k = 0; k < i; k++)
    {
        s += tab[k];
    }
    if (s > n)
    {
        return false;
    }
    int occ[i];
    for (int k = 0; k < i; k++)
    {
        occ[k] = 0;
    }
    for (int k = 0; k < i; k++)
    {
        occ[tab[k]]++;
    }
    for (int k = 0; k < i; k++)
    {
        if (occ[k] > tab[k])
        {
            return false;
        }
    }
    return true;
}
```

Q42– Quelle est la réponse à la grande question, la vie, l'univers et le reste ?

🛟 Indication : la réponse est dans la question !

La réponse est 42, c'est une référence habituelle en informatique à la série de science-fiction *Le Guide du voyageur galactique* de Douglas Adams. Voir la la page wikipedia pour plus de détails