DS 2

Informatique de tronc commun, classe de PC

Julien Reichert

Ce devoir consiste en deux parties de programmation et une partie sur des bases de données totalement indépendantes. Les fonctions et requêtes à écrire ne sont pas forcément de difficulté croissante.

Partie 1 : Bases de données « tournois de Skat »

On considère la base de données suivante, formée de quelques tables (adaptéees) issues de la base de données du site de la fédération française de Skat :

- `TOURNOI`, avec les attributs `id` (clé primaire, entier), `titre` (clé, chaîne de caractères), `organisateur` (clé étrangère vers une table nommée `UTILISATEURS` qui ne sera pas utilisée dans ce sujet, entier), `date` (chaîne de caractères), `club` (clé étrangère vers une table nommée `CLUB` qui ne sera pas utilisée non plus, entier), `series` (entier), `etat` (entier) ;
- `INSCRIPTION`, avec les attributs `id` (clé étrangère vers la table `TOURNOI`, entier), `numero` (entier), `nom` (chaîne de caractères), `utilisateur` (clé étrangère vers la table `UTILISATEURS`, entier), `parti` (entier) ;
- `SERIE`, avec les attributs `id` (clé étrangère vers la table `TOURNOI`, entier), `serie` (entier), `numero` (correspondant à l'attribut de même nom dans la table `INSCRIPTION`, entier), `numtable` (entier), `place` (entier), `resultat` (entier), `gagnes` (entier), `perdus` (entier) ;

Explications de certains attributs :
- `series` : nombre de séries que le tournoi durera et donc nombre d'entrées de la table `SERIE` par joueur inscrit (sauf si la valeur de son attribut `parti` n'est pas à `0` mais à un entier `n`, auquel cas le nombre de séries auquel il aura participé est `n`) ;
- `etat` : le numéro de la série en cours (si le tournoi est fini, c'est le nombre de séries plus un) ;
- `numero` : numéro d'inscription d'un joueur pour le tournoi en cours, **en recommençant à un pour chaque tournoi** (mais un retrait avant le démarrage du tournoi peut entraîner des trous dans la numérotation) ;
- `utilisateur` : ce champ permet d'associer des résultats à un compte en vue de faire des classements ;
- `parti` : déjà expliqué ci-avant ;
- `serie` : nécessairement entre `1` et la valeur de l'attribut `series` du tournoi associé ;
- `numtable` : nécessairement entre `1` et le nombre de joueurs participant à la série divisé par `4`, arrondi par excès, les trois ou quatre joueurs ayant la même valeur pour cet attribut à une série fixée jouent donc ensemble ;
- `place` : nécessairement entre `1` et `4`, sans trou dans la numérotation à série et table fixées ;
- `resultat` : score du joueur dans la série en question ;
- `gagnes` : nombre de jeux gagnés dans la série en question (analogue pour `perdus`).

Exercice 1.1 : Justifier que `(id, numero)`, `(id, nom)` et `(id, utilisateur)` sont des clés pour la table `INSCRIPTION` et que `(id, serie, numero)` et `(id, serie, numtable, place)` sont des clés pour la table `SERIE`.

Exercice 1.2 : Écrire une requête affichant le nombre de joueurs ayant participé au « Tournoi de Nancy ».

Exercice 1.3 : Écrire une requête affichant le nombre de points du joueur « Julien Reichert » au tournoi numéro `5`.

Exercice 1.4 : Écrire une requête affichant le plus grand score obtenu dans une série d'un tournoi organisé dans le club numéro `2`.

Exercice 1.5 : Écrire une requête affichant le nombre de points du vainqueur du tournoi numéro `2`.

Exercice 1.6 : Écrire une requête affichant l'identifiant de l'utilisateur ayant marqué le plus de points sur l'ensemble des tournois. En cas d'égalité, un seul résultat suffit.

Exercice 1.7 : Écrire une requête affichant la liste des joueurs qui vont participer à la série numéro `n` du tournoi numéro `t` dans l'ordre décroissant de leur classement actuel (pour être rigoureux, le classement est déterminé selon ces critères dans l'ordre : le total des points, le total des jeux gagnés et le plus faible nombre de jeux perdus).

Exercice 1.8 : Écrire une requête affichant la liste des joueurs ayant participé au plus grand nombre de tournois, en précisant ce nombre de tournois en tant que deuxième attribut de chaque enregistrement affiché.

Partie 2 : Autour de la suite de Syracuse

On considère la (en pratique un pluriel s'imposerait sans doute) suite de Syracuse, définie pour un u_0 quelconque par la relation de récurrence

$$u_{n+1} = \begin{cases} \frac{u_n}{2} & \text{si } u_n \text{ est pair} \\ 3u_n + 1 & \text{sinon.} \end{cases}$$

La conjecture de Collatz (portant également d'autres noms) s'énonce ainsi : « Pour tout entier naturel non nul u_0, existe-t-il un rang n pour lequel $u_n = 1$? ». Cette conjecture n'est toujours pas démontrée à l'heure actuelle. Le but de cette partie est de faire quelques exercices autour de la suite de Syracuse.

Exercice 2.1 : Calculer à la main les termes de la suite avec $u_0 = 19$ jusqu'à atteindre 1. Pour information, ce sera en u_{20}.

On définit plusieurs valeurs associées à un entier u_0 :

- Le temps de vol est le premier indice n tel que $u_n = 1$. Ce sera donc 20 pour l'exemple.
- Le temps de vol en altitude est le premier indice n tel que $u_n < u_0$.
- L'altitude maximale est le maximum de la suite (en admettant que 1 soit atteint).

Exercice 2.2 : Pour la suite avec $u_0 = 19$, préciser le temps de vol en altitude et l'altitude maximale. Par ailleurs, que dire du temps de vol en altitude si u_0 est pair, et comment se comporte la suite après que 1 a été atteint ?

Pour le calcul informatique des valeurs ci-avant, dans certains cas il est pertinent de le faire indépendamment, mais dans d'autres mieux vaut profiter d'autres valeurs de u_0 en faisant appel à la mémoïzation.

Exercice 2.3 : Commencer par écrire une fonction calculant le terme suivant de la suite en prenant le terme actuel en argument. On utilisera cette fonction dans les exercices suivants.

Exercice 2.4 : Écrire une fonction prenant en argument un entier `N` supposé strictement supérieur à `1` et renvoyant le temps de vol en altitude de la suite de Syracuse avec u_0 valant `N`.

Exercice 2.5 : Écrire une fonction prenant en argument un entier `N` supposé strictement supérieur à `1` et renvoyant la liste des temps de vol de toutes les suites de Syracuse avec u_0 compris entre `1` et `N`. Il est attendu que l'on ne gaspille pas de la complexité en temps par le calcul de l'intégralité des suites en question. L'approche top-down est attendue vu l'impossibilité de faire un bottom-up a priori. La structure de mémoire intermédiaire pourra au choix être une liste ou un dictionnaire.

Exercice 2.6 : Écrire une fonction prenant en argument un entier `N` supposé strictement supérieur à `1` et renvoyant la liste des altitudes maximales de toutes les suites de Syracuse avec u_0 compris entre `1` et `N`. Les consignes sont les mêmes que pour l'exercice précédent.

Partie 3 : Recherche de facteur dans un texte

La recherche de facteur dans un texte correspond au problème suivant : on dispose de deux chaînes de caractères `str` et `mot`, cette dernière étant de taille notée `k`, et on demande s'il existe un indice `i` tel que la sous-chaîne formée par les caractères aux indices de `i` inclus à `i + k` exclu de `str` forme exactement `mot`. En particulier, `k` sera inférieur ou égal à la taille de `str` et en pratique il sera même négligeable.

Ce problème se décline en plusieurs variantes : déterminer le premier indice `i` correspondant, déterminer le nombre de tels indices (avec chevauchements possibles pour les sous-chaînes égales à `mot`), ou déterminer la liste de tous les indices correspondants.

Dans toute cette partie, on autorise la notation `t[a:b]` pour construire la tranche de `t` allant des indices `a` inclus à `b` exclu, et on rappelle que la complexité de cette opération est linéaire en la taille de la tranche construite.

Exercice 3.1 : Écrire une fonction résolvant le problème de la recherche simple de facteur dans un texte (donc renvoyant un booléen).

Exercice 3.2 : Calculer la complexité en temps de la fonction précédente dans le pire des cas.

Exercice 3.3 : Pour gagner du temps dans le cas moyen, on remplacera l'extraction d'une tranche par une boucle conditionnelle lors de la vérification qu'un indice `i` convient pour le problème. Réécrire la fonction de l'exercice 3.1 sans utiliser la notation fournie ci-avant (si ce n'est pas déjà fait).

Exercice 3.4 : Calculer la complexité de la fonction précédente dans le meilleur des cas.

Exercice 3.5 : Détailler l'exécution de la fonction de l'exercice 3.3 (éventuellement 3.1 si l'exercice 3.3 n'est pas traité) lorsque `str = "CHERCHER CHEZ CHER"` et `mot = "CHEZ"`.

En vue d'optimiser la complexité dans le cas moyen, nous présentons ici deux algorithmes (au programme de la MP2I), dans une version simplifiée : l'algorithme de Boyer-Moore et l'algorithme de Rabin-Karp.

Pour l'algorithme de Boyer-Moore, le principe est de ne pas tester tous les indices de la grande chaîne comme départs potentiels du mot, en fonction du résultat des essais précédents. Pour ceci, deux particularités sont ajoutées à la recherche naïve : d'une part, la comparaison caractère par caractère se fait depuis la fin de la petite chaîne (comparée à la fin de la zone de la grande chaîne ainsi comparée), et d'autre part une liste de dictionnaires permet de déterminer en fonction de l'indice où un échec a eu lieu et du caractère de la grande chaîne trouvé au lieu du caractère attendu (celui qui correspondait dans la petite chaîne) de combien on décale l'indice de départ.

Concrètement, on imagine la petite chaîne comme un cadre qu'on essaie de superposer sur la grande chaîne, et si cela échoue, on déplace le cadre à droite jusqu'à trouver un moyen d'aligner la lettre dont la comparaison a échoué.

La construction de la liste de dictionnaires se fait en étudiant la petite chaîne seule, et un exemple est donné en annexe. Une version simplifiée revient à faire ceci : on crée un dictionnaire vide par indice de la petite chaîne, on insère la clé correspondant au premier caractère de la petite chaîne au dictionnaire d'indice zéro, associé à la valeur zéro, puis pour chaque indice à partir d'un, on prend chaque clé du dictionnaire à l'indice précédent et on la reporte dans le dictionnaire à l'indice actuel, en ajoutant un à la valeur par rapport à ce qui précède, puis on insère (on remplace en cas d'existence) la clé correspondant au caractère de l'indice actuel dans la petite chaîne, associée à la valeur zéro.

Le pseudo-code de l'algorithme de Boyer-Moore, dans sa version permettant de compter le nombre de fois où la petite chaîne figure dans la grande chaîne est par ailleurs fourni en annexe (il appelle la fonction `precalcule` qui est à écrire sans pseudo-code à partir des indications).

Exercice 3.6 : Écrire la fonction de pré-calcul de la liste de dictionnaires.

Exercice 3.7 : Écrire une version de l'algorithme de Boyer-Moore qui **détermine si** la petite chaîne figure dans la grande chaîne. On pourra s'inspirer du pseudo-code fourni.

Concernant l'algorithme de Rabin-Karp, le principe est le même que pour la recherche naïve, mais à chaque étape, avant de commencer à comparer les caractères de la petite chaîne (notée ici `m`) et de l'extrait de la grande chaîne (extrait noté ici `ts`), une fonction de hachage est utilisée pour déterminer si les hachés de `ts` et de `m` coïncident. Si ce n'est pas le cas, on passe à l'extrait commençant un indice plus loin.

Tout l'intérêt de l'algorithme revient à calculer rapidement l'image de chaque extrait. Moralement, il ne faudrait pas que ce soit en temps linéaire en la taille de l'extrait, comme le ferait le calcul, sans utiliser d'astuce, des fonctions de hachage classiques à base d'évaluation polynomiale.

Concrètement, on considère que la fonction de hachage, appelée sur une chaîne (que ce soit la petite chaîne intégralement ou un extrait de la grande chaîne) se définit mathématiquement ainsi, avec des paramètres constants choisis ici arbitrairement :

$$f(s_0 s_1 \dots s_{k-1}) = (\sum_{i=0}^{k-1} s_i 97^{k-1-i}) \bmod 997$$

en assimilant chaque caractère à son code dans la table de caractères ASCII (en Python, on utilise `ord` pour passer d'un caractère à son code).

Alors pour passer de l'image de $s_i s_{i+1} \dots s_{i+k-1}$ à l'image de $s_{i+1} s_{i+2} \dots s_{i+k}$, il suffit de multiplier par 97, de retrancher $s_i \times 97^k$ et d'ajouter s_{i+k} puis de calculer le reste modulo 997. On évitera de ruiner l'optimisation en calculant 97^k une fois pour toutes.

Exercice 3.8 : Écrire une fonction reprenant le principe ci-avant, ayant pour arguments la chaîne `s`, l'indice de départ `i` d'un extrait de cette chaîne, le haché `h` de cet extrait, la taille `k` de l'extrait et la variable `c` valant `97**k`, et renvoyant le haché de l'extrait de `s` de même taille et démarrant en `i+1`.

Exercice 3.9 : Écrire une fonction calculant le haché d'une chaîne (à utiliser pour le premier extrait ainsi que pour la petite chaîne). La complexité doit être linéaire en la taille de la chaîne (mais inutile de le prouver).

Exercice 3.10 : Écrire une fonction réalisant l'algorithme de Rabin-Karp.

Annexe : Exemple de table de redirection pour la petite chaîne `"CARACTERE"`. Les lettres du mot permettent de mieux voir les choses et ne figurent pas dans le programme. En colonnes figurent les clés de chaque dictionnaire et dans les cellules les valeurs (si la cellule est vide, la clé est absente, et les valeurs `OK` et `WIN` correspondent à zéro, là aussi pour être plus explicite). À côté de la table de redirection figure un pseudo-code de Boyer-Moore **qui renvoie le nombre de fois** où `m` figure dans `s`.

En bleu figurent les nombres qui pourraient être optimisés dans une version plus avancée de l'algorithme.

Clés des dictionnaires :	'A'	'C'	'E'	'R'	'T'
Dictionnaire à l'indice 0 ('C')		WIN			
Dictionnaire à l'indice 1 ('A')	OK	+1			
Dictionnaire à l'indice 2 ('R')	+1	+2		OK	
Dictionnaire à l'indice 3 ('A')	OK	+3		+1	
Dictionnaire à l'indice 4 ('C')	+1	OK		+2	
Dictionnaire à l'indice 5 ('T')	+2	+1		+3	OK
Dictionnaire à l'indice 6 ('E')	+3	+2	OK	+4	+1
Dictionnaire à l'indice 7 ('R')	+4	+3	+1	OK	+2
Dictionnaire à l'indice 8 ('E')	+5	+4	OK	+1	+3

```
fonction boyer_moore(s, m) {
  tab <- precalcule(m);
  n <- taille(s); k <- taille(m); i <- 0; reponse <- 0;
  tant que i < n - k + 1 {
    j <- k-1;
    tant que j > -1 et s[i+j] = m[j] décrémenter j;
    si j = -1 alors {
      incrémenter reponse; incrémenter i;
    }
    sinon si s[i+j] est une clé de tab[j] alors
      augmenter i de tab[j][s[i+j]];
    sinon augmenter i de j+1;
  }
  renvoyer reponse;
}
```