R1.01 : Initiation au développement (partie 2)
Feuille TD n° 3

Algorithmes classiques pour d'éléments homogènes : Recherches dans un tableau

Objectifs :
1.- Savoir appliquer un modèle d'algorithme connu.
2.- Savoir analyser un problème et identifier un modèle connu d'algorithme participant à sa résolution
3.- Sensibilisation à la notion de complexité temporelle

EXERCICE 1 - Rechercher une valeur dans un tableau ordonné d'entiers

Etant donnés :
- `tab` un tableau de `lgTab` valeurs entières, *ordonné par ordre* ***décroissant***, c'est à dire tel que `tab(i) ≥ tab(i+1)` pour tout `0 ≤ i < lgTab-1`
- et une valeur entière `val`

On souhaite écrire un sous-programme `recherchePremiereOcc` qui indique si (oui ou non) `val ∈ tab`, et si oui, donne sa position dans `tab`.

Partie 1 –Déclaration et Appel

Travail à faire

1. La position de `val` dans `tab` est-elle unique ? Dans ce cas, décrire cette position. Dans quel cas elle pourrait être unique ? ***Argumentez - Expliquez***.
2. Écrire la *déclaration C++* du sous-programme `recherchePremiereOcc` en respectant les noms qui vous ont déjà été fournis (nom du sous-programme et des éléments `tab`, `lgTab` et `val`).
3. Dans le sous-programme `main()` écrire *l'appel* de `recherchePremiereOcc`, en vue de rechercher le contenu de `valCherchee` dans un tableau `monTab` trié par ordre décroissant strict. Compléter le code avec *les déclarations* des éléments que vous jugez utiles, *et l'action exploitant* les résultats de l'appel.

```
1  int main ()
2  {
3      // ELEMENTS DU PROGRAMME (constantes et variables)
4      const unsigned short int TAILLE = 10;
5      int monTab [TAILLE] = {60, 45, 30, 25, 15, 10, 0, -15, -20, -45};
6                            // trié strictement décroissant
7      int valCherchee;      // valeur cherchée dans monTab
8      …
9
10     // TRAITEMENTS
11     …
12     // … >> Initialisation valeur cherchee >> valCherchee
13     …
14
15     // ?  >> rech. 1ere occ. >> ??
16     …
17
18
19     // ???  >> exploiter résultat recherche >> (écran)
20     …
21
22     return 0;
23 }
```

Code n° 1 : Appel de `recherchePremiereOcc` à compléter

Partie 2 – Stratégie de l'algorithme : basée sur le modèle du parcours dichotomique

Condition d'application (pré-condition)

La recherche dichotomique est applicable grâce aux 2 propriétés vérifiées par le tableau utilisé :

- Structure à accès direct
- Tableau trié

Rappel du Principe

- Le parcours dichotomique consiste à diviser par 2, à chaque itération, la portion de tableau restant à parcourir.
  Nous appellerons respectivement `borneInf` et `borneSup` la borne inférieure (respectivement borne supérieure) de la portion de tableau restant à parcourir
- A chaque itération, l'élément courant à analyser est celui situé au **milieu** de l'espace de recherche restant à parcourir. Cet indice du milieu est un entier calculé par la formule suivante : `(borneInf + borneSup)/2`, où la division est une division entière.

Travail à faire

4. Pour mieux comprendre le principe du parcours dichotomique, reporter sur le tableau l'évolution des **éléments** de l'algorithme lors de la recherche dichotomique de la valeur 3 dans le tableau d'entiers ci-dessous.

0	1	2	3	4	5	6	7	8	9
60	45	30	25	15	10	0	-15	-20	-45

val	borneInf	borneSup	milieu
3	0	9	

Tableau 1 : Tableau de suivi d'exécution de la recherche dichotomique pour un tableau et une valeur cherchée donnés

5. Exprimer les conditions de fin d'itération à l'aide des éléments de l'algorithme
6. Écrire l'algorithme correspondant à cette stratégie, accompagné des spécifications internes succinctes nécessaires.

Partie 3 – Élaboration d'un jeu d'essai permettant de tester le programme et Calcul des performances de l'algorithme

7. Établir le jeu d'essai qui vous permettra de tester et valider le bon fonctionnement de l'algorithme de recherche dichotomique. Il sera composé des tableaux que vous jugerez pertinents d'utiliser, et des valeurs recherchées pertinentes pour chaque tableau.
8. Sensibilisation à la notion de complexité : **à faire en TP**
   a) Compléter cet algorithme de sorte qu'il calcule (et affiche) le nombre de fois qu'une case du tableau sera accédée au cours de l'exécution de l'algorithme.
   b) Est-ce que ce nombre est toujours le même, quelle que soit la valeur cherchée ?
   c) Donner alors :
      - le nombre minimum d'accès nécessaires pour trouver un élément dans le tableau
      - le nombre maximum d'accès nécessaires pour trouver l'élément dans le tableau. S'appuyer sur le cours : il dit que le nombre maximum d'itérations pour aboutir à une zone de recherche contenant 1 seule case est $\leq \log_2(n)$, où n est la taille initiale du tableau
      - le nombre moyen d'accès nécessaires pour trouver un élément dans le tableau

   d) Comparer ces valeurs avec les mêmes valeurs obtenues lorsque la recherche est séquentielle

Partie 4 – Tableau d'enregistrements : à faire en TP

Supposons maintenant que le tableau **`tab`** contient **`lgTab`** enregistrements décrivant des personnes, et que l'on souhaite lancer la recherche d'une personne par son **nom**.

Chaque enregistrement est du type **`UnePersonne`** suivant :

```
struct UnePersonne
{
    string nom;
    string prenom;
    UneAdresse adresse;
};
```

avec

```
struct UneAdresse
{
    string numRue;
    string nomRue;
    unsigned short int codePostal;
    string nomVille;
};
```

Travail à faire

9. Quelles sont les pré-conditions (conditions sur les données de l'algorithme) pour pouvoir appliquer la recherche de première occurrence développée à la question précédente ?
10. En supposant toutes les conditions satisfaites, adapter l'algorithme à ce cas particulier de recherche.

EXERCICE 2
Déterminer la première/dernière des occurrences dans un tableau ordonné d'entiers

Soit `tab` un tableau d'entiers *ordonné* par valeurs ***décroissantes***, c'est à dire tel que `tab(i)` ≥ `tab(i+1))` pour tout `0` ≤ `i` < `lgTab-1`. On accepte donc les doublons.
On souhaite écrire un sous-programme **`determinerPremierDernier`** qui indique, pour un entier `val` donné, si `val` ∈ `tab` et si oui, donne la position dans `tab`, respectivement :

- de l'occurrence de `val` située le plus à gauche dans `tab`
- de l'occurrence de `val` située le plus à droite dans `tab`

Travail à faire

1. Donner les résultats fournis par ce sous-programme pour les tableaux suivants, où `lgtab` = 10

	0	1	2	3	4	5	6	7	8	9	
tab	60	60	60	60	60	60	60	60	60	60	et `val` = 60
tab	60	50	45	40	30	20	20	20	10	5	et `val` = 20
tab	60	50	45	40	30	20	20	20	10	5	et `val` = 5
tab	60	50	45	40	30	20	20	20	10	5	et `val` = 14

2. Écrire la déclaration C++ de ce sous-programme en respectant les noms qui vous ont déjà été fournis (nom du sous-programme et des éléments `tab`, `lgTab` et `val`).

3. Compléter le programme **`main()`** ci-dessous avec *l'appel* du sous-programme **`determinerPremierDernier`**, les déclarations des éléments que vous jugez utiles, et l'action exploitant les résultats produits par cet appel.

```
1  int main (void)
2  {
3      // ELEMENTS DU PROGRAMME (constantes et variables)
4      …
5      const unsigned short int TAILLE = 10;
6      int monTab [TAILLE] = {60, 50, 45, 40, 30, 20, 20, 20, 10, 5}; // décroissant
7      int valCherchee;    // valeur cherchée dans monTab
8      …
9
10     // TRAITEMENTS
11     …
12     // … >> Initialisation valeur cherchee >> valCherchee
13     …
14
15
16     // ? >> determinerPremierDernier  >> ??
17     …
18
19     // ???  >> exploiter résultat >> (écran)
20     …
21
22     return 0;
23 }
```

Code n° 2 : Appel de `determinerPremierDernier` à compléter

4. ***Définir la stratégie adoptée*** : Indiquer le(s) modèle(s) d'algorithme qui sera/seront utilisé(s) pour écrire son algorithme. Pour se rapprocher des standards, votre stratégie devra combiner des modèles d'algorithmes connus. ***Argumentez / précisez*** votre proposition.

5. Écrire l'algorithme accompagné des spécifications internes succinctes nécessaires.