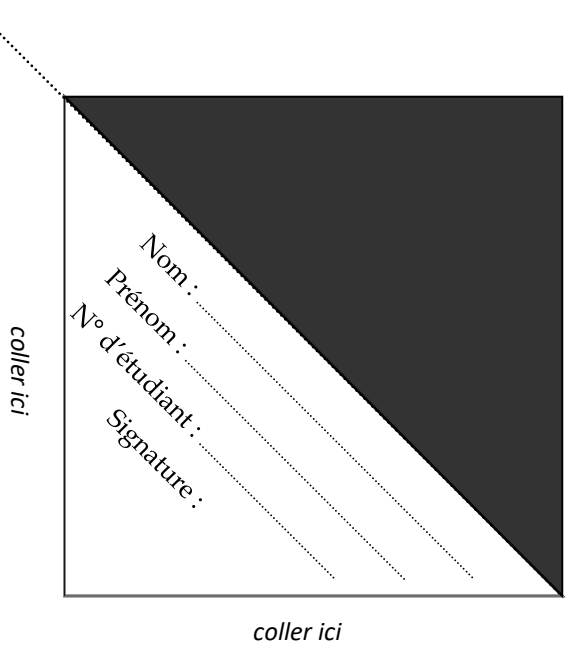
Université Claude Bernard Lyon 1

Année universitaire : 2016 / 2017

LIFAP3 : Algorithmique et programmation avancée

Contrôle mi-parcours
20 mars 2017
Durée : 1h30

Note : / 20



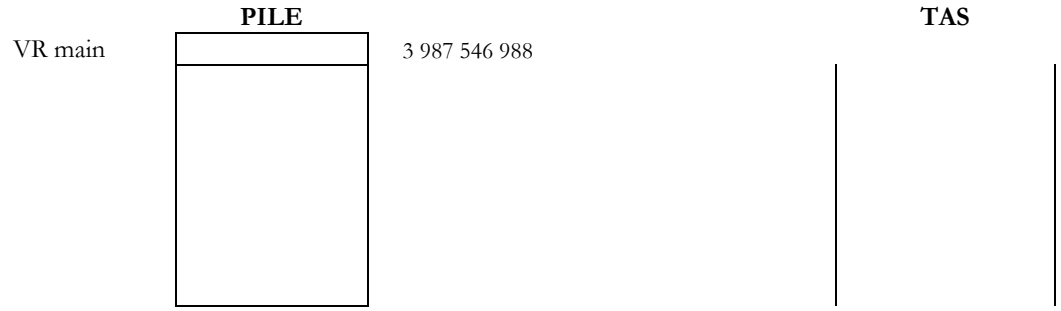
Documents et téléphones portables interdits. Le barème est donné à titre indicatif. Répondez dans les emplacements prévus à cet effet. Travaillez au brouillon d'abord de sorte à rendre une copie propre – nous ne pouvons pas vous garantir une copie supplémentaire. **Il sera tenu compte de la présentation et de la clarté de vos réponses**.

Exercice 1 : Trace mémoire (6 points)

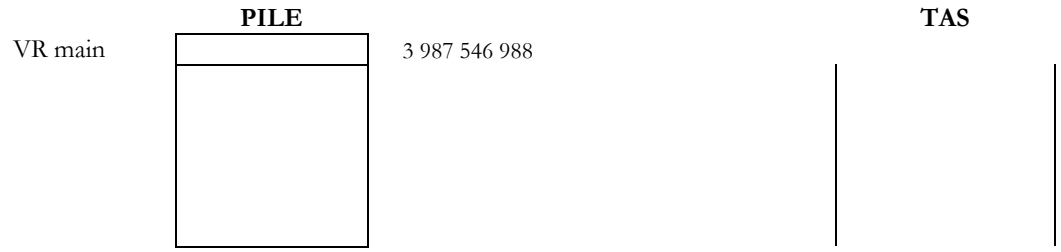
Question 1.1 : Supposons la classe **Etudiant** et le programme principal donnés ci-dessous. Dessiner l'état de la mémoire aux deux endroits indiqués en commentaires, et donner la trace écran. Vous utiliserez le modèle théorique de pile vu en cours et en TD. Vous supposerez que la valeur de retour du main est stockée à l'adresse 3 987 546 988.

```
1   #include <iostream>
2   using namespace std;
3
4   class Etudiant {
5   public:
6      int numero;
7      char * prenom;
8
9      Etudiant (int n, char * p) {numero = n; prenom = p;}
10     void afficher () const {
11        cout << "Prenom : " << prenom << "\nNumero : " << numero << endl;
12     }
13     char initiale () const {
14        /* Dessinez l'état de la mémoire #1 */
15        return prenom[0];
16     }
17  };
18
19  int main () {
20     Etudiant * e = new Etudiant(100,"Jean");
21     e->afficher();
22     if (e->initiale() == 'J') {
23        char * nvPrenom = "Denis";
24        e->prenom = nvPrenom;
25     }
26     /* Dessinez l'état de la mémoire #2 */
27     return 0;
28  }
```

Etat de la mémoire #1



Etat de la mémoire #2



Trace écran :

Question 1.2 : Citer deux choses qui ne vont pas dans ce code.

1

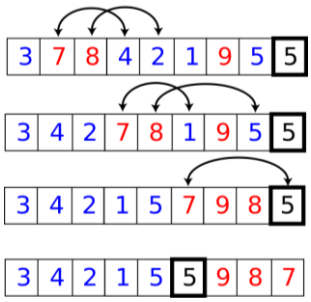
2.

Exercice 2 : Tri rapide (9 points)

Dans cet exercice, nous allons étudier l'algorithme de tri dit rapide (quick sort). Cette méthode de tri consiste à placer un élément du tableau (appelé pivot) à sa place définitive, en permutant tous les éléments de telle sorte que tous ceux qui sont inférieurs au pivot soient à sa gauche et que tous ceux qui sont supérieurs au pivot soient à sa droite. Cette opération s'appelle le partitionnement. Pour chacun des sous-tableaux (le gauche et le droit), on définit un nouveau pivot et on répète l'opération de partitionnement. Ce processus est répété récursivement, jusqu'à ce que l'ensemble des éléments soit trié.

Concrètement, pour partitionner un **sous-tableau** :

- on définit le pivot comme étant le dernier élément (valeur 5 ci-contre)
- on déplace tous les éléments inférieurs ou égaux au pivot vers le début du tableau (étapes 1 et 2 ci-contre)
    - en échangeant chaque élément plus petit ou égal au pivot avec un élément plus grand (celui le plus à gauche)
- on place le pivot à la fin des éléments déplacés (étapes 3 et 4 ci-contre)
    - en échangeant avec la première valeur plus grande que le pivot



Question 2.1 : Compléter la procédure suivante qui échange deux éléments du tableau donnés par leurs indices.

```
void echangeValeur (int * tab, unsigned int indice1, unsigned int indice2) {




}
```

Question 2.2 : Traduire la fonction **`partitionner`** suivante en C++. Cette fonction partitionne le sous-tableau entre les indices premier et dernier, et elle retourne l'indice de la valeur pivot à la fin du partitionnement.

Fonction partitionner (tab : tableau d'entiers, premier : entier positif, dernier : entier positif) : entier positif
Préconditions : le tableau est initialisé, premier et dernier sont tous deux compris entre 0 et la taille du tableau - 1
Postconditions : le tableau est partitionné
Résultat : l'indice de la valeur pivot à la fin du partitionnement
Paramètres en mode donnée : premier et dernier
Paramètres en mode donnée-résultat : tab
Variables locales : i, j : entiers
Début
  j ← premier
  Pour i allant de premier jusqu'à dernier-1 par pas de 1 faire
    Si tab[i] ≤ tab[dernier] alors
      echangeValeur(tab,i,j)
      j ← j + 1
    Fin si
  Fin pour
  echangeValeur(tab,j,dernier)
  retourne j
Fin partitionner

Question 2.3 : Donner la complexité en notation O de cet algorithme en fonction de n le nombre d'éléments entre les indices premier et dernier. Vous n'avez pas besoin de compter le nombre exact d'opérations réalisées mais vous devez justifier votre réponse.

Question 2.4 : Compléter la procédure récursive **`triRapideRecursif`** suivante qui trie le tableau en paramètre entre les indices premier et dernier.
Indication : trier le tableau consiste à le partitionner en deux sous-tableaux autour d'un pivot et de trier récursivement les deux sous-tableaux.

```
void triRapideRecursif (int * tab, unsigned int premier, unsigned int dernier) {









}
```

Question 2.5 : Donner la complexité en notation O de cette procédure récursive en fonction de n le nombre d'éléments entre les indices premier et dernier. Vous n'avez pas besoin de compter le nombre exact d'opérations réalisées mais vous devez justifier votre réponse.

Question 2.6 : Ecrire l'entête et le code C++ de la procédure **`triRapide`** qui trie un tableau passé en paramètre. La taille du tableau est aussi passée en paramètre.

Exercice 3 : Classe et complexité (5 points)

Dans cet exercice, on souhaite associer un tableau de réels en simple précision avec sa taille dans une classe que l'on nommera **`Tableau`**. Cette classe contiendra les membres suivants :

- Une donnée membre **`tab`** qui contient l'adresse mémoire du premier élément du tableau
- Une donnée membre **`taille`** valant le nombre d'éléments du tableau
- Un constructeur qui initialise un tableau vide, et un destructeur qui libère la mémoire allouée sur le tas
- Des fonctions membres de manipulation du tableau (non précisées ici)

Question 3.1 : Donner le code C++ de la classe **`Tableau`** (données membres, constructeur et destructeur).

Soit la fonction membre **appartient** suivante donnée en notation algorithmique :

```
    Fonction appartient (Tableau t) : booléen
    Résultat : vrai si le tableau t apparaît dans le tableau de
    l'instance, faux sinon
    Paramètre en mode donnée : t
    Variables locales : i : entier
    Début
1     i ← 0
2     Tant que i ≤ taille – t.taille faire
3         Si egal(t,i,i+t.taille-1) alors
4             retourne vrai
5         FinSi
6         i ← i + 1
7     Fin tant-que
8     retourne faux
    Fin partitionner
```

Vous supposerez connue la fonction membre **egal** qui teste l'égalité entre le tableau **t** passé en paramètre et le sous-tableau de l'instance entre les indices donnés en paramètre. La fonction **egal** fait **t.taille** comparaisons de réels.

Question 3.2 : Donner le nombre de comparaisons de réels faites lors de l'exécution de la fonction **appartient** dans le pire des cas, en fonction de la taille du tableau en paramètre et la taille du tableau de l'instance. En conclure sur la complexité de la fonction.