R3.02 – Développement efficace

TP 0 *(associé au Cours 0)*

Récursivité

(1 séance encadrée)

Pour ce TP télécharger et compéter le projet CLion fourni.

Exercice 1 : Mise en jambe

I. Fonction factorielle (observation)

Pour information, la fonction factorielle est fournie.

```
int factorielleRec(const int n)
// {n ≥ 0} => {résultat = n! (soit 1 × 2 × … × (n − 1) × n}
```

On fournit aussi une version de la fonction factorielle qui retourne, par effet de bord grâce au paramètre **`cpt`** passé par référence, le nombre de multiplication effectué pour produire le résultat du calcul.

```
int factorielleRecNbMults(const int n, int& cpt)
// {n ≥ 0} => {résultat = n! (soit 1 × 2 × … × (n − 1) × n ;cpt permet de
compter le nombre de multiplications effectuées}
```

II. Procédure Hanoï

Implanter et tester la procédure **`hanoi`** récursive à laquelle vous avez réfléchi en cours :

```
void hanoi(int nbDisques, char depart, char intermediaire, char arrivee)
// {n ≥ 1} => {affiche les déplacements à effectuer pour résoudre le pb}
```

Implanter et tester la procédure **`hanoiNbDeplacements`** qui calcule aussi le nombre de déplacement effectués. Après avoir vérifié qu'elle compte correctement le nombre de déplacements sur quelques exemples, commenter les lignes d'affichage de déplacement.

Observer alors le nombre de déplacements effectués pour un nombre de disques compris entre 1 et 10 et proposer la formule générale pour un nombre de disque quelconque n.

```
void hanoiNbDeplacements (int nbDisques, char depart, char intermediaire,
char arrivee, int& cpt)
// {n ≥ 1} => {affiche les déplacements à effectuer pour résoudre le
pb ; cpt permet de compter le nombre de déplacements effectués pour
résoudre le problème}
```

Exercice 2 : Fonction puissance

I. Fonction puissance : force brute (application directe de la définition récursive)

x puissance n (avec $x \in \mathbb{R}$ et $n \in \mathbb{N}$), notée x^n, se définit récursivement naturellement comme suit :

$$puissance(x,n) = \begin{cases} 1, & n = 0 \\ x \times puissance(x, n-1), & n > 0 \end{cases}$$

Implanter et tester la fonction récursive naturelle suivante avec **x** = 5 et **n** = 0, 1, 2, 4, 8, 16, … :

```
double puissance(float x, int n)
// {} => {résultat = x puissance n }
```

Écrire maintenant la fonction suivante qui a un paramètre résultat qui fournit le nombre de multiplications effectuées (ce doit être $n-1$) :

```
double puissanceNbMults(float x, int n, int& nbmult)
// {} => {résultat = x puissance n, nbmult est le nombre total de
multiplications effectuées lors du calcul}
```

II. Fonction puissance : diviser pour régner

Dans une approche diviser pour régner la fonction puissante peut s'exprimer comme suit :

$$\text{puissanceDR}(x,n) = \begin{cases} 1, & n = 0 \\ x, & n = 1 \\ \text{puissanceDR}\left(x, \frac{n}{2}\right) \times \text{puissanceDR}\left(x, \frac{n}{2}\right), & n > 1 \text{ et } n \text{ pair} \\ x \times \text{puissanceDR}\left(x, \frac{n}{2}\right) \times \text{puissanceDR}\left(x, \frac{n}{2}\right), & n > 1 \text{ et } n \text{ impair} \end{cases}$$

où $n/2$ est le quotient entier de la division de n par 2.

Implanter la fonction suivante de **manière efficace** pour minimiser le nombre de multiplications (la tester avec les mêmes valeurs que la fonction `puissanceNbMult`, qu'observe-t-on ?) :

```
double puissanceDRNbMult(float x, int n, int& nbmult)
// {} => {résultat = x puissance n, nbmult est le nombre total de
multiplications effectuées lors du calcul}
```

Note : vous devez obtenir un nombre de multiplications égal à $\log_2 n$, sinon c'est que votre programme n'est pas efficace, il faut mieux réfléchir !

Exercice 3 : Procédure min max

Implanter la procédure itérative suivante :

```
template<class T>
void minMaxIterNbComp(vector<T>& v, T& min, T& max, int& nbComp)
// {v.size()≥ 1} => {min = plus petite valeur de v, max = plus grande
valeur de v, nbcomp = nombre de comparaisons impliquant au moins un
élément de v}
```

que vaut `nbComp` ? (vous devriez trouver $2(n-1)$ où n est le nombre d'éléments de **v**)

Réfléchir à une implantation sous la forme d'un algorithme récursif en diviser pour régner qui résout le même problème :

```
template<class T>
void minMaxDRNbComps(vector<T>& v, T& min, T& max, int& nbComp)
// {v.size()≥ 1} => {min = plus petite valeur de v, max = plus grande
valeur de v, nbcomp = nombre de comparaisons impliquant au moins un
élément de v}
```

INDICES

- la BASE est constituée de 2 situations triviales (quelles sont les tailles de vecteurs pour lesquelles la solution au problème est immédiate ?)
- la RECURRENCE : si on résout problème sur deux sous vecteurs de **v**, comment fusionner les résultats obtenus pour avoir le résultat sur le vecteur **v** ?

que vaut **nbComp** ? (vous devriez trouver une valeur inférieure à celle produite par **minMaxIter**, et plus exactement $\frac{3}{2}n - 2$ où n est le nombre d'éléments de **v**)

Si cela vous intéresse, vous pouvez essayer d'implanter la version itérative de la procédure **minMaxDR** qui fait exactement le même nombre de comparaisons, mais en utilisant un algorithme itératif cette fois.

Exercice 4 : Autour de l'approche diviser pour régner

Avec mes collègues qui interviennent dans le module, nous avons élaboré la stratégie de tri suivante pour trier un vecteur **v** entre une borne inférieure **inf** et une borne supérieure **sup** :

- si la taille du vecteur est inférieure ou égale à 6 faire un tri par insertion de **v** entre **inf** et **sup** (la BASE, ou la situation triviale)
- sinon (la RECURRENCE)
  1) trouver, parmi les valeurs **v[inf+sup/2]**, **v[inf]** et **v[sup]**, la valeur **valPartage** du médian des 3 (on a un plus petit, le médian (**valPartage**) et un plus grand),
  2) ranger le plus petit à la position **inf**, le plus grand à la position **sup** et **valPartage** à la position **sup-1**,
  3) ensuite, parcourir **v[inf..sup-2]** pour ranger en tête les éléments inférieurs à **valPartage** et en fin les éléments supérieurs à **valpartage** en repérant l'indice (**indiceSep**) de séparation
  4) on va alors permuter **v[indiceSep]** et **v[sup-1]** (c'est **valPartage**), ce qui a pour effet de déjà mettre **valPartage** à la place qu'il occupera quand le vecteur **v** d'origine sera complètement trié
  5) on va ensuite trier récursivement deux vecteurs : **v[inf..indiceSep-1]** (trier les plus petits que **valPartage**) et **v[indiceSep+1..sup]** (trier les plus grands que **valPartage**) ; c'est fini !

Cette stratégie est celle d'un worker récursif qui fait le travail, à savoir :

```
template<class T>
void triR302Worker(vector<T>& v, int inf, int sup)
```

La procédure de tri fera simplement l'appel du worker comme suit :

```
template<class T>
void triR302(vector<T>& v) {
  triM3103Worker(v, 0, v.size()-1);
}
```

Faire des dessins pour comprendre comment ça marche, et implanter successivement les étapes 1) à 5) comme indiqué ci-dessous.

Implanter une procédure qui réalise les étapes 1) et 2), elle peut avoir l'entête suivante :

```
template<class T>
T partage(vector<T>& v, const int inf, const int sup)
// {v.size()≥ 1} => {résultat = valeur du médian tel que défini dans
l'étape 1) et mise en place du plus petit et du plus grand dans v telles
que définies dans l'étape 2)}
```

Réfléchir à une version itérative de l'étape 3)

Les étapes 4) et 5) sont très faciles.

Implanter maintenant et tester le triR302, pour aller plus vite vous pouvez chercher sur le Web une procédure de tri par insertion.

Le squelette du tri est le suivant :

```
template<class T>
void triR302Worker(vector<T>& v, int inf, int sup)
  if (inf+6 > sup) {
    triInsertion(v, inf, sup);  // cas de base
                                // trouver une version sur le Web
  } else {
    int valPartage = partage(v, inf, sup);  // 1) & 2)
    // 3)
    // 4)
    // 5)
  }
}
```

Lorsque vous avez terminé vous pouvez modifier la procédure de tri pour compter le nombre de comparaisons effectué par le tri. Vous devriez trouver une valeur inférieure à n^2 et proche de $n.\log_2 n$. Où n est la taille du vecteur trié.