Algorithmique Avancé

TP 2

Récursivité, Liste et Complexité

Ce TP a pour but déterminer la complexité d'un algorithme dans un premier temps puis d'implémenter des algorithmes récursifs.

1 Récursivité

Un algorithme récursif est un algorithme qui fait appel à lui même.
"C'est tout ? Bah c'est pas si compliqué, allez salut."
"Hopopop ptit malin, c'est pas aussi simple alors ramène toi."
La méthode récursive est souvent un autre moyen de voir un problème. Plutôt que de passer par une boucle itératif tel qu'un ***for*** ou un ***while***, on appel le même algorithme avec des paramètres différents pour répéter plusieurs fois les mêmes instructions mais dans un contexte qui évolue.
Exemple :

Algorithm 1 Recursive Sum

```
function SUM(n : entier)
    Si n > 1 Alors
        Retourner n + SUM(n − 1)
    fin Si
    Retourner n
fin function
```

Cet algorithme fait la somme de 1 jusqu'à n. On peut voir le problème ainsi :

$$sum = \sum_{1}^{n}$$

$$sum = \sum_{1}^{n-1} +n$$

$$sum = \sum_{1}^{n-2} +(n-1) + n$$

$$...$$

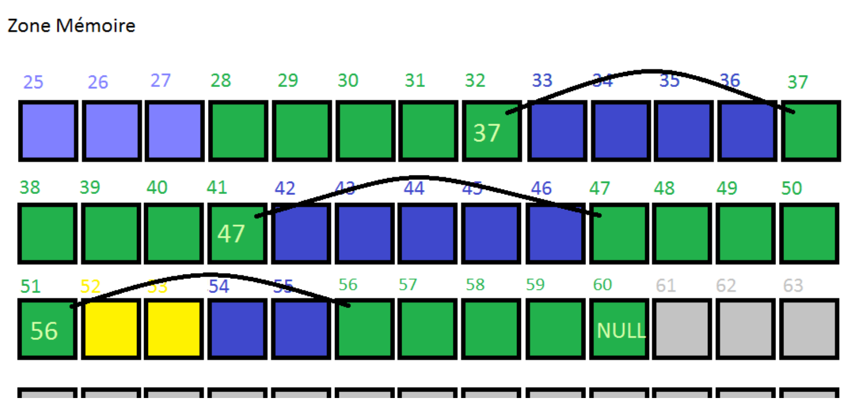
On obtient ainsi une récursion. Pour résoudre le problème au niveau n, il faut résoudre le problème au niveau inférieur.
Attention : Pour concevoir un algorithme récursif qui fonctionne, il faut penser :

— À la condition d'arrêt, une condition qui va retourner un résultat sans faire appel à la fonction, pour éviter une récursion infinie
— À l'itération, un changement dans les paramètres lors de l'appel de la fonction, pour éviter une récursion infini

Il est facile d'arriver à un "Stack Overflow" lorsqu'on implémente une fonction récursive. Pensez bien à ces deux points lors de votre implémentation.

2 Liste, Pile et File

Un **tableau** en `C/C++` se traduit par la réservation d'un emplacement mémoire assez grand pour y stocker l'ensemble des données que le tableau va stocker. Cet emplacement mémoire est segmenté en plusieurs petites cases qui se suivent, cela permet d'accéder rapidement à une case donnée, il suffit d'ajouter l'indice de cette case à l'adresse de la première case. L'inconvénient en revanche c'est l'aspect statique de la structure, une fois la taille défini, redéfinir la taille est couteux. Les **listes chaînées** sont là pour répondre au besoin d'avoir une quantité de données dynamiques.
Une **liste chaînée** est une façon de stocker un ensemble de données qui consiste à disperser les données sur des emplacements mémoires discontinues et de rajouter un élément dans ces zones mémoires qui permet de retrouver les autres (comme un pointeur en `C/C++`).



2.1 Pile/File ou LIFO/FIFO

Les **piles** et les **files** sont plus des façons de traiter des données plutôt que des façons de les stocker.
Une **pile**(peu importe sa façon de stocker) va restreindre l'accès aux données en ne donnant que la dernière valeur qu'il a stocké. Une fois la dernière valeur stocké, il dépile les données, permettant ainsi de donner accès à l'avant-dernière valeur stockée. C'est un **LIFO**, **L**ast *I*n *F*irst **O**ut. Une **file**, lui, va restreindre l'accès aux données en ne donnant que la première valeur qu'il a stocké. Une fois la première valeur stocké, il dépile les données, permettant ainsi de donner accès à l'avant-dernière valeur stockée. C'est un **FIFO**, **F**irst *I*n *F*irst **O**ut.

3 Complexité d'un algorithme

3.1 Keskesé ?

Pour mesurer l'efficacité d'un programme on mesure à quelle il résout un problème. Cependant, les ordinateurs exécute les programmes à des vitesses différentes, il arrive même qu'un ordinateur exécute le même programme à des vitesses différentes. On mesure donc cette vitesse en déterminant le nombre d'instructions exécutées selon le nombre d'élément à traiter.
Prenons l'algorithme suivant :

Algorithm 2 Sum of square odd

t $\Leftarrow$ tableau de n nombre aléatoire
sum $\Leftarrow$ 0
Pour chaque valeur *v* de *t* **faire**
  Si v est impaire **Alors**
    sum $\Leftarrow v \times v$
  fin Si
fin Pour

Pour un tableau de n nombre on effectue un test et une multiplication $\Rightarrow$ on effectue donc $2 \times n$ instructions.
Si $f(n)$ est le nombre d'instruction étant donné un nombre n d'élément, on note la **complexité d'un programme** $O(g(n))$ tel qu'il existe $C > 0$ et $n_0 > 0$ pour lesquels $f(n) \leq C.g(n)$. Pour faire simple il s'agit d'un **ordre de grandeur**.
Dans notre exemple a une complexité en $O(n) \Leftrightarrow f(n) = 2n, g(n) = n, C = 3, n_0 = 0$

3.2 Exemple

Algorithm 3 Polynome evaluation

```
coeff ⇐ tableau de n coefficient
powerValues ⇐ tableau de n puissances
x ⇐ abscisse du point
sum ⇐ 0
Pour chaque indice i de coeff faire
    poweredX ⇐ x
    Pour j allant de 1 à powerValues[i] faire
        poweredX ⇐ poweredX × x
    fin Pour
    sum ⇐ sum + coeff[i] × poweredX
fin Pour
```

Cet algorithme évalue un polynome en un point, supposons que les puissances soient rangées dans l'ordre allant de 1 à n. La puissance de x nécessite 1 instruction, puis 2, ... et ainsi de suite jusqu'à n. Le nombre d'instruction est donc égale à :

$$f(n) = \sum_{i=0}^{n} +2n$$

$$f(n) = \frac{n(n+1)}{2} + 2n$$

L'ordre de grandeur est de n^2, la complexité de ce programme est donc $O(n^2)$
Une version améliorer du programme multiplierai la puissance précédente avec x pour obtenir la puissance courante :

Algorithm 4 Better polynome evaluation

```
coeff ⇐ tableau de n coefficient
powerValues ⇐ tableau de n puissances
x ⇐ abscisse du point
sum ⇐ 0
poweredX ⇐ 1
Pour chaque indice i de coeff faire
    poweredX ⇐ poweredX × x
    sum ⇐ sum + coeff[i] × poweredX
fin Pour
```

Avec cette amélioration on obtient un nombre instruction égale à $3n \Rightarrow$ La complexité est donc de $O(n)$.

3.3 Notations

Le nombre d'instruction peut varier d'une exécution à un autre. On mesure la complexité O en determinant le nombre d'instruction maximum autrement-dit le pire cas. Mais on peut aussi calculer le nombre d'instructions moyen qu'on note Θ et minimum Ω

3.4 Temps d'exécution

La plupart des algorithmes qui traitent un ensemble d'élément peuvent être classés selon leur temps d'exécution :

Constant : Complexité en $O(1)$, le nombre d'instruction reste le même peut importe le nombre d'élément à traiter. Exemple : la structure de données $std :: unordered_map$ en C++ permet de chercher et d'insérer des éléments en un temps constant. Pour chaque élément du tableau, la structure enregistre une clé d'indexage qui permet de déterminer l'adresse mémoire où l'élément est rangé.

Logarithmique : Complexité en $O(log_2(n))$, l'algorithme continue tant que le nombre d'élément à traiter est divisible par 2. Exemple : La recherche dichotomique, on regarde où se situe l'élément à chercher par rapport à la moitié tableau et on réitère cette recherche dans la partie inférieur ou supérieur du tableau jusqu'à trouver l'élément.

Linéaire : Complexité en $O(n)$. Exemple : Recherche Séquentielle, on teste chaque case du tableau pour trouver notre élément.

Quasi-linéaire : Complexité en $O(nlog_2(n))$. Exemple : Le tri rapide ou tri par fusion.

Polynomial : Complexité en $O(n^k)$. Exemple : Le tri à bulle avec une complexité de $O(n^2)$

Exponentiel rapide : Complexité en $O(c^{log(n)})$.

Exponentiel : Complexité en $O(c^n)$.

Factoriel : Complexité en $O(n!) \equiv O(n^n)$.

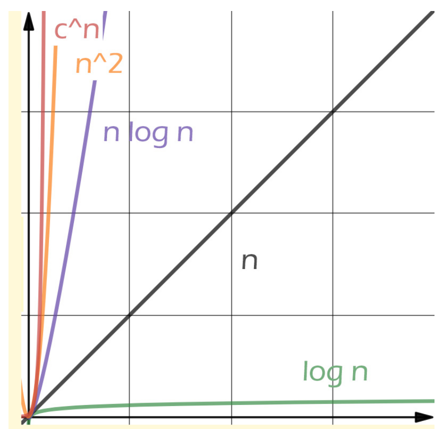


FIGURE 1 – Complexities

4 TP

4.1 Programmation

Le dossier *Algorithme_TP1/TP* contient un dossier *C++*. Vous trouverez dans ce dossier des fichiers *exo<i>.pro* à ouvrir avec *QtCreator*, chacun de ces fichiers projets sont associés à un fichier *exo<i>.cpp* à compléter pour implémenter les différentes fonctions ci-dessus. Le fichier *exo0.cpp* est un exemple d'implémentation de fonction récursive.

Notes :

— L'objet $Context$ est présent au début des fonctions récursives afin de d'enregistrer l'appel de la fonction et d'afficher l'appel dans la fenêtre.

— L'instruction $return\_and\_display$ permet d'enregistrer la fin de la fonction et d'afficher le résultat de la fonction. Vous pouvez toutefois utiliser l'instruction $return$ normal.

— La classe $QVector2D$ est implémenté dans le framework Qt pour décrire un point.

```
QVector2D point;
float x = point.x(); // return the component x of point
float y = point.y(); // return the component y of point
point.setX(float value); // set a new value to the component x
point.setY(float value); // set a new value to the component y
point.length(); // return the module/length of point
```

Le type $Array$ implémente des fonctions d'accès et de modifications de tableau tel que get(), insert(), set()...

```
void sort(Array& toSort)
{
  int firstNumber = toSort.get(0); // get the first number of the array toSort
  int lastNumber = toSort.get(toSort.size()-1); // get the last number of the
      array toSort
  if (firstNumber > lastNumber)
    toSort.swap(0, toSort.size()-1); // swap between the first index and the last
  printf("I think it's sorted");
}
```

```
  Array array(int size); // Array array(10) --> make an array of 10 numbers
      intialized to -1;
  array.get(int index); // array.get(2) --> get the number at index 2
  array[int index]; // equivalent to get()
  array.set(int index, int value); // array.set(2, 10) --> set 10 into the 2nd
      case of array
  array.swap(int index1, int index2); // array.swap(2,5) --> swap the 2nd and the
      5th case of array
  array.insert(int index, int value); // array.set(2, 10) --> insert 10 into the 2
      nd case of array by shifting the all next numbers
```

Implémenter les fonctions suivantes à l'aide d'un algorithme récursif :

- **power**(int $value$, int n) : retourne la $n$ième puissance de $value$
- **fibonacci**(int n) : retourne la $n$ième valeur de Fibonacci
- **search**(int $value$, int $array$[], int size) : retourne l'index de $value$ dans $array$

- **allEvens**(int $evens[]$, int $array[]$, int $evenSize$, int $arraySize$) : rempli $evens$ avec tous les nombres paires de $array$. $evenSize$ représente le nombre de valeur dans $evens$ (est donc égale à 0 au début) et $arraySize$ est le nombre de valeur dans $array$.
- **mandelbrot**(vec2 z, vec2 point, int n) : retourne vrai si le point appartient à l'ensemble de Mandelbrot pour la fonction $f(z) \rightarrow z^2 + point$. Un point appartient à cet ensemble si la suite z_n est bornée, autrement-dit s'il existe un $i < n$ tel que $|z_i| < 2$.

Implémenter une structure DynaTableau et une structure Liste avec les comportements suivants :
- **ajoute**(int $valeur$) : Ajoute une nouvelle valeur à la fin de la structure (alloue de la mémoire en plus si nécessaire)
- **recupere**(int n) : Retourne le $n$ième entier de la structure
- **cherche**(int $valeur$) : Retourne l'index de $valeur$ dans la structure ou -1 s'il n'existe pas
- **stocke**(int n, int $valeur$) : Redéfini la $n$ième valeur de la structure avec $valeur$

Ajouter des fonctions à la structure de votre choix pour implémenter le comportement d'une Pile et d'une File
- **pousser_file**(int $valeur$) : Ajoute une valeur à la fin ou au début de la structure
- **retirer_file**() : Enlève la première valeur ajoutée et la retourne
- **pousser_pile**(int $valeur$) : Ajoute une valeur à la fin ou au début de la structure
- **retirer_pile**() : Enlève la dernière valeur ajoutée et la retourne

4.2 Exercices

Déterminer la complexité minimum Ω et maximum O des algorithmes suivantes :

Algorithm 5 Insertion Sort

t $\Leftarrow$ tableau de n nombre aléatoire
sorted $\Leftarrow$ tableau de n -1
Pour chaque indice *i* de *t* **faire**
    insertionIndex $\Leftarrow 0$
    Tant que *sorted[insertionIndex]* ≥ 0 et *t[i]* $\geq$ *sorted[insertionIndex]* **faire**
        insertionIndex $\Leftarrow$ *insertionIndex* $+ 1$
    fin Tant que
    sorted.insert(*insertionIndex*, *t[i]*)
fin Pour

Algorithm 6 String Distance

s1 $\Leftarrow$ chaine de n caractère
s2 $\Leftarrow$ chaine de m caractère
i $\Leftarrow 1$
distance $\Leftarrow 0$
Tant que $i < m - 1$ et $i < n - 1$ **faire**
    cost1 $\Leftarrow abs($*s1[i]* $-$ *s2[i]*$)$
    cost2 $\Leftarrow abs($*s1[i]* $-$ *s2[i - 1]*$)$
    cost3 $\Leftarrow abs($*s1[i]* $-$ *s2[i + 1]*$)$
    distance $\Leftarrow$ *distance* $+ min($*cos1*, *cos2*, *cos3*$)$
fin Tant que

Algorithm 7 Binary Search

t $\Leftarrow$ tableau de n nombre aléatoire triés
toSearch $\Leftarrow$ nombre à chercher
start $\Leftarrow$ 0
end $\Leftarrow n$
Tant que *start* $<=$ *end* **faire**
    mid $\Leftarrow \frac{start+end}{2}$
    Si *toSearch* $> t[mid]$ **Alors**
        start $\Leftarrow$ *mid*
    Sinon Si *toSearch* $< t[mid]$ **Alors**
        end $\Leftarrow$ *mid*
    Sinon
        foundIndex $\Leftarrow$ *mid*
    fin Si
fin Tant que

Algorithm 8 Search All

t $\Leftarrow$ tableau de n nombre aléatoire triés
toSearch $\Leftarrow$ nombre à chercher
start $\Leftarrow 0$
end $\Leftarrow n$
Tant que *start* $<=$ *end* **faire**
    mid $\Leftarrow \frac{start+end}{2}$
    Si *toSearch* $>$ *t[mid]* **Alors**
        start $\Leftarrow$ *mid*
    Sinon
        end $\Leftarrow$ *mid*
    fin Si
fin Tant que
Si *t[start+1]* $==$ *toSearch* **Alors**
    iMin $\Leftarrow$ *start+1*
    i $\Leftarrow$ *iMin*
    Tant que *t[i]* $==$ *toSearch* **faire**
        i $\Leftarrow i + 1$
    fin Tant que
    iMax $\Leftarrow i - 1$
Sinon
    iMin $\Leftarrow -1$
    iMax $\Leftarrow -1$
fin Si

Algorithm 9 Binary Sort

t $\Leftarrow$ tableau de n nombre aléatoire
sorted $\Leftarrow$ tableau vide
Pour chaque indice *i* de *t* **faire**
    sorted.insert(binarySearch(*sorted*, *t[i]*), *t[i]*)
fin Pour