IMAC

TP Algorithmique 2

Complexité et Récursivité

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

1 Complexité d'un algorithme

1.1 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 1 Sum of square odd

```
t ⇐ tableau de n nombre aléatoire
sum ⇐ 0
Pour chaque valeur v de t faire
    Si v est impaire Alors
        sum ⇐ v × 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 $O(n) \Leftrightarrow f(n) = 2n, g(n) = n, C = 3, n_0 = 0$

1.2 Exemple

Algorithm 2 Polynome evaluation

coeff $\Leftarrow$ tableau de n coefficient
powerValues $\Leftarrow$ tableau de n puissances
x $\Leftarrow$ abscisse du point
sum $\Leftarrow$ 0
Pour chaque *indice* i de *coeff* **faire**
    poweredX $\Leftarrow$ *x*
    Pour *j* allant de 1 à *powerValues[i]* **faire**
        poweredX $\Leftarrow$ *poweredX* $\times$ *x*
    fin Pour
    sum $\Leftarrow$ *sum* $+$ *coeff[i]* $\times$ *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 3 Better polynome evaluation

coeff $\Leftarrow$ tableau de n coefficient
powerValues $\Leftarrow$ tableau de n puissances
x $\Leftarrow$ abscisse du point
sum $\Leftarrow$ 0
poweredX $\Leftarrow$ 1
Pour chaque *indice* i de *coeff* **faire**
    poweredX $\Leftarrow$ *poweredX* $\times$ *x*
    sum $\Leftarrow$ *sum* $+$ *coeff[i]* $\times$ *poweredX*
fin Pour

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

1.3 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

1.4 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 $O(n!) \equiv O(n^n)$.

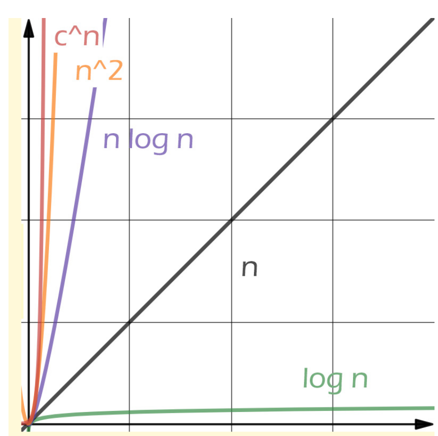

Figure 1 – Complexities

1.5 Exercices

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

Algorithm 4 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 5 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 6 Binary Search

```
t ⇐ tableau de n nombre aléatoire triés
toSearch ⇐ nombre à chercher
start ⇐ 0
end ⇐ n
Tant que start <= end faire
    mid ⇐ (start−end)/2
    Si toSearch > t[mid] Alors
        start ⇐ mid
    Sinon Si toSearch < t[mid] Alors
        end ⇐ mid
    Sinon
        foundIndex ⇐ mid
    fin Si
fin Tant que
```

Algorithm 7 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 8 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

2 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 9 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 :

$$
\begin{aligned}
sum &= \sum_{1}^{n} \\
sum &= \sum_{1}^{n-1} + n \\
sum &= \sum_{1}^{n-2} + (n-1) + n \\
&\ldots
\end{aligned}
$$

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.

3 TP

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 $array$.
- **mandelbrot**(vec2 z, vec2 point, int n) : retourne vrai si le point appartient à l'ensemble de Mandelbrot pour la fonction $f(z) \to z^2 + c$
- **quickSort** du TP1
- **mergeSort** du TP1

Vous pouvez utiliser le langage que vous souhaitez.

3.1 C++

Le dossier *Algorithme_TP2/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 :

— La macro $NOTIFY\_START$ est appelé pour afficher dans la fenêtre l'appel courante.

— L'instruction $return$ a été modifié pour dire à la fenêtre que la fonction courante se termine. Vous devez utiliser des parenthèses pour la valeur de retour et utiliser les $\{\}$ si le $return$ se trouve dans un if même s'il n'y a qu'une ligne.