Algorithmique et programmation 2
Synthèse de cours

Juin 2022

Sommaire

1 Introduction

1.1 Présentation du document

Ce document est une synthèse du cours d'Algorithme et Programmation 2 faite par un étudiant. Dans ce document, on peut y retrouver deux ressources principales : les algorithmes et les temps d'exécution. Bien évidemment, le cours d'Algorithme ne se résume pas qu'à ces deux points. Mais ce document peut s'avérer utile pour les révisions quand on sait que dans tous les sujets, il y a un QCM où les questions portent souvent sur le temps d'exécution et que l'on peut nous demander d'écrire mot pour mot un algorithme dans la partie où l'on doit coder. Ce document n'a pas pour but d'être lu entièrement.

Cette synthèse de cours commence par une partie expliquant quelques points sur le pseudo-code utilisé ce semestre car cela m'avait quelque peu perturbé lorsque j'étais en L1. Ensuite arrivent les parties sur les algorithmes et les temps d'exécution. On peut voir qu'au total, il y a une trentaine d'algorithmes à apprendre. Ce n'est donc pas une matière qui se révise la veille, mais qui doit se travailler *régulièrement*.

1.2 Banalités

Lors de mon année, au partiel, il me semble que la moyenne de la promo était de 5 ou 6. En outre, à la fin du semestre, il y avait 108 étudiants éligibles au rattrapage d'Algorithme et Programmation 2. À titre de comparaison, 79 étudiants étaient éligibles au rattrapage d'Algorithme et Programmation 1. C'est donc environ 30 étudiants de plus, soit une classe. Il ne faut donc surtout pas négliger cette matière. Il est important de faire l'*effort* d'apprendre par cœur les algorithmes et les temps d'exécution assez tôt de sorte à ne pas accumuler du retard. La *régularité* est la clé pour une telle quantité de travail, surtout que les cours du semestre 2 sont beaucoup plus denses qu'au semestre dernier.

Lors d'une épreuve notée, et surtout en Algorithme et Programmation 2, évitez de négliger les derières notions que vous avez vues en vous disant qu'elles ne tomberont pas. Lors de mon partiel, il y avait une question sur l'invariant de boucle (*loop invariant*) de `Heapsort` et peu de personnes ont su répondre à cette question étant donné que la plupart avait négligé cette partie.

Pour préparer les QCM, n'hésitez surtout pas à refaire ceux sur Moodle car le professeur peut parfois remettre exactement la même question. Faites attention avec le QCM car il est à point négatif. Un bon nombre d'étudiants avait eu 0 au QCM à cause des malus. Cela peut faire très mal. Avec 3 réponses correctes et 3 fausses, on obtient 1,5/6 au QCM.

Notez par ailleurs que sur Python, il peut y avoir au maximum 1000 appels récursifs. Au-delà, l'interpréteur Python renvoie un erreur. Il est important de retenir le nombre de 1000 car dans les QCM, il est déjà arrivé que la question porte sur l'affichage d'un algorithme avec moins de 1000 appels récursifs (par exemple 354) mais aussi avec plus (par exemple 2581).

2 Pseudo-code du S2

2.1 Définitions

Défintion : code
On appelle code la description d'un algorithme dans un *langage de programmation* (Python, C, etc.)

Définition : pseudo-code
On appelle pseudo-code la description d'un algorithme dans un *langage naturel* (français, anglais, etc.) Cette description suit une syntaxe et des conventions particulières. Cependant, les règles de syntaxe et les conventions varient en fonction de l'enseignant. Il n'existe pas un unique pseudo-code français. De même, il n'existe pas un unique pseudo-code anglais.

Remarque :
Dans cette synthèse de cours, on appellera alors pseudo-code du S1, et non pseudo-code français, le pseudo-code utilisé en Algorithmique et Programmation au semestre 1 pour ne pas insinuer qu'il existe un seul pseudo-code français. Similairement, on appellera pseudo-code du S2, et non pseudo-code anglais, le pseudo-code utilisé en Algorithmique et Programmation au semestre 2.

2.2 Particularités du pseudo-code du S2

Le pseudo-code du S1 est différent de celui du S2. Par exemple, au semestre 2, on ne déclare pas le type des variables dans le pseudo-code contrairement au semestre 1. Une manière de comprendre le pseudo-code du S2 est de se dire qu'il est identique au code en Python à quelques différences près. Voici une liste de ces différences :

- En Python, on utilise le mot-clé `elif` pour exprimer un sinon si, mais dans le pseudo-code du S2, on utilise le mot-clé `else if`.
- Contrairement au code en Python, quand on déclare une fonction, on n'écrit pas le mot-clé `def` et on ne met pas les deux-points.
- Plus globalement, on ne met jamais les deux-points, pas même dans une boucle for ou while, ou dans une instruction conditionnelle `if`, `else if` ou `else`.
- Dans un tableau, l'indice du premier élement est 1 dans le pseudo-code du S2 alors que c'est 0 en Python. Ainsi, si on pose `A=[1,2]`, dans le pseudo-code du S2, `A[1]` référence la valeur 1 et en Python, `A[1]` référence la valeur 2.
- Pour créer un tableau `L` de taille n, on écrit `Let L[1..n] be new array`. Notez qu'il n'y a que deux petits points entre 1 et n, et non trois petits points.
- Pour créer deux tableaux `L` et `R` de taille respectif n et m dans la même ligne, on écrit `Let L[1..n] and R[1..m] be new arrays`.
- Si l'on veut écrire `POUR i ALLANT_DE 1 A 100`, on écrit `for i=1 to n`.
- Si l'on veut écrire `POUR i ALLANT_DE 100 A 1`, on écrit `for i=100 down to 1`.
- Lorsque l'on écrit `for i=1 to n`, aussi bizarre que cela puisse paraître, `i` prend successivement les valeurs de `1, 2, ..., n-1, n` et `n+1` ! En revanche, le bloc d'instructions de la boucle est exécuté pour `i=1, i=2, ..., i=n` mais pas pour `i=n+1`. En fait, dans `for i=1 to n`, à la dernière itération, `i` prend la valeur de `n+1`, mais on ne rentre pas

dans la boucle car à chaque nouvelle valeur de `i`, on effectue le test `i<=n`. Pour bien comprendre, on peut voir `for i=1 to n` comme un équivalent de l'instruction dans le langage de programmation C `for(i=1;i<=n;i++)`. Décrivons cette instruction. Tout d'abord, `i` prend la valeur de 1, puis on teste que `i<=n`. Supposons que ce soit le cas, alors on rentre dans la boucle. Une fois les instructions de la boucle exécutées, on passe à la deuxième itération de la boucle. Ainsi, on exécute l'instruction `i++`, c'est-à-dire qu'on incrémente `i` de 1. On teste que `i<=n`. Si c'est le cas, on rentre dans la boucle. Puis, on exécute l'instruction `i++`. On teste que `i<=n`. Si c'est le cas, alors on rentre dans la boucle, etc. À un moment, `i=n`. On teste que `i<=n`. C'est le cas donc on rentre dans la boucle. Enfin, on exécute l'instruction `i++` (`i` vaut alors `n+1`). On teste que `i<=n`. Ce n'est pas le cas donc on ne rentre pas dans la boucle. Notez alors la subtilité suivante : l'instruction `for i=1 to n` est executée $n+1$ fois. Mais, le bloc d'intructions associé à `for i=1 to n` est exécutée n fois car on ne rentre pas dans la boucle pour `i=n+1`. Cette information peut être utile pour les QCM.

- Pour affecter à `k` la valeur `1`, on écrit `k` $\leftarrow$ `1`, mais `k=1` est accepté.
- Pour effectuer une comparaison large entre `a` et `b`, on écrit `a` $\leq$ `b` (respectivement `a` $\geq$ `b`), mais `a<=b` (respectivement `a>=b`) est accepté.
- En Python, les tirets sont interdits pour les noms des fonctions. Si on le fait, l'interpréteur Python nous renvoie une erreur. En revanche, dans le pseudo-code du S2, on a le droit d'utiliser un tiret pour nommer une fonction. Ainsi, on peut appeler une fonction `Insertion-Sort`.
- Si l'on pose `x=1` et `y=2`, pour échanger les valeurs de `x` et `y`, on peut écrire `Exchange x with y`. Notez que l'on écrit `with` et non `and`. Notez aussi qu'écrire `Exchange y with x` revient à la même chose.
- Les valeurs $+\infty$ et $-\infty$ existent dans le pseudo-code du S2, et on peut effectuer des comparaisons avec ces deux valeurs. Notez que la proposition `x` $< +\infty$ est vraie pour tout `x` réel. Notez aussi que les comparaisons avec $+\infty$ et $-\infty$ sont utiles pour la conception de certains algorithmes.
- Si l'on veut signaler qu'il y a une erreur, on écrit `Error "Le texte à afficher"`.
- On a le droit de faire appel à des constantes ou des fonctions mathématiques sans les déclarer tant qu'on utilise les notations conventionnelles. Par exemple, on peut écrire `c=`$\lfloor\pi\rfloor$ sans préciser que la fonction $x \mapsto \lfloor x \rfloor$ est la fonction partie entière et que π est la constante égale à 3,141592... Mais on ne peut pas écrire `c=pe(cstArc)` car ce n'est pas conventionnellement admis que `pe` correspond à la fonction partie entière et que `cstArc` correspond à π, aussi connu sous le nom de constante d'Archimède.
- Soit A un tableau. En Python, `len(A)` correspond à la taille du tableau A. Dans le pseudo-code du S2, on note `A.lenght` la taille du tableau A.
- Il n'est pas nécessaire d'écrire `2*i` ou `2`$\times$`i`. On peut simplement écrire `2i` en pseudo-code du S2.
- Pour effectuer un test d'égalité entre `a` et `b`, on écrit `a==b` mais pour faire un test de non-égalité, on écrit `a` $\neq$ `b`.
- La valeur `None` de Python s'écrire `NIL` en pseudo-code du S2. Il s'agit de l'abréviation du mot latin *nihil* signifiant *rien* ou *le vide*.

Ensuite, voici quelques règles pour confirmer la ressemblance entre le pseudo-code du S2 et le code en Python :

- Pour effectuer une comparaison stricte entre `a` et `b`, on écrit `a < b` ou `a > b`.
- Pour retourner la valeur `x` dans une fonction, on écrit `return x`.
- Après avoir effectué une déclaration de fonction, de boucle ou d'instruction conditionnelle, on effectue une indentation.
- Si l'on veut écrire `SI x=1 ALORS`, on écrit `if x=1`.
- Si l'on veut écrire `SINON`, on écrit `else`.
- Si l'on veut écrire `TANT_QUE i < n FAIRE`, on écrit `while i < n`.

3 Algorithmes

Voici une liste de tous les algorithmes vus dans cette matière. Ils sont tous à apprendre.

3.1 Insertion-Sort

```
Insertion-Sort(A,n)
    for j = 2 to n
        key <- j
        i <- j-1
        while i>0 and A[i]>key
            A[i+1] <- A[i]
            i <- i-1
        A[i+1] <- key
```

3.2 Merge

```
Merge(A,p,q,r)
    n1 <- q-p+1
    n2 <- r-q
    Let L[1..n1+1] and R[1..n2+1] be new arrays
    for i=1 to n1
        L[i] <- A[p+i-1]
    for j=1 to n2
        R[j] <- A[q+j]
    L[n1+1] <- +∞
    R[n2+1] <- +∞
    i <- 1
    j <- 1
    for k=p to r
        if L[i] ≤ R[j]
            A[k] <- L[i]
            i <- i+1
        else
            A[k] <- R[j]
            j <- j+1
```

3.3 Merge-Sort

```
Merge-Sort(A,p,r)
    if p < r
    q <- ⌊(p+r)/2⌋
    Merge-Sort(A,p,q)
    Merge-Sort(A,q+1,r)
    Merge(A,p,q,r)
```

3.4 Parent

```
Parent(i)
    return ⌊i/2⌋
```

3.5 Left

```
Left(i)
    return 2i
```

3.6 Right

```
Right(i)
    return 2i+1
```

3.7 Max-Heapify

```
Max-Heapify(A,i)
    l <- Left(i)
    r <- Right(i)
    if l ≤ A.heapsize and A[l] > A[i][1]
        largest <- l
    else
        largest <- i
    if r ≤ A.heapsize and A[r] > A[largest]
        largest <- r
    if largest ≠ i
        Exchange A[i] with A[largest]
        Max-Heapify(A,largest)
```

[1] L'ordre des conditions est très très *important*. Dans tous les langages de programmation, les conditions sont lues linéairement, c'est-à-dire que si l'on a `if (condition 1) and (condititon 2)`, on vérifie d'abord la condition 1 puis la condition 2. Pour comprendre l'intérêt de cette remarque, considérons l'instruction, `if l ≤ A.length and A[l] > A[i]` avec `l = A.length+1`. Dans un premier temps, on vérifie que `l ≤ A.length`. Ce n'est pas le cas donc *on ne vérifie pas la deuxième condition* (très important) et on n'exécute pas le bloc d'intructions associé au `if`. Maintenant, considérons l'instruction `if A[l] > A[i] and l ≤ A.length` avec `l = A.length+1`. On vérifie tout d'abord `A[l] > A[i]`. Pour faire cela, on évalue `A[l]`, c'est-à-dire `A[A.length+1]`. Or cette case n'existe pas. On aurait le message d'erreur *list assignment index out of range* indiquant que l'on dépasse l'intervalle autorisée des indices.

3.8 Build-Max-Heap

```
Build-Max-Heap(A)
    A.heapsize <- A.length
    for i = ⌊A.length/2⌋ down to 1
        Max-Heapify(A,i)
```

3.9 Heapsort

```
Heapsort(A)
    Build-Max-Heap(A)
    for i=A.length down to 2
        Exchange A[1] with A[i]
        A.heapsize <- A.heapsize-1
        Max-Heapify(A,1)
```

3.10 Quicksort

```
Quicksort(A,p,r)
    if p < r
        q <- Partition(A,p,r)
        Quicksort(A,p,q-1)
        Quicksort(A,q+1,r)
```

3.11 Partition

```
Partition(A,p,r)
    x <- A[r]
    i <- p-1
    for j=p to r-1
        if A[j] ≤ x
            i <- i+1
            Exchange A[i] with A[j]
    Exchange A[i+1] with A[r]
    return i+1
```

3.12 Randomized-Partition

```
Randomized-Partition(A,p,r)
    i <- Random(p,r)
    Exchange A[i] with A[r]
    return Partition(A,p,r)
```

3.13 Randomized-Quicksort

```
Randomized-Quicksort(A,p,r)
    if p < r
        q <- Randomized-Partition
        Randomized-Quicksort(A,p,q-1)
        Randomized-Quicksort(A,q+1,r)
```

3.14 Minimum

```
Minimum(A)
    n <- A.length
    min <- A[1]
    for j=2 to n
        if A[j] < min
            min <- A[j]
    return min
```

3.15 Maximum

```
Maximum(A)
    n <- A.length
    max <- A[1]
    for j=2 to n
        if A[j] > max
            max <- A[j]
    return max
```

3.16 Randomized-Select

```
Randomized-Select(A,p,r,i)
    if p==r
        return A[p]
    q <- Randomized-Partition(A,p,r)
    k <- q-p+1
    if i==k
        return A[q]
    else if i < k
        return Randomized-Select(A,p,q-1,i)
    else
        return Randomized-Select(A,q+1,r,i-k)
```

3.17 Heap-Maximum

```
Heap-Maximum
    return A[1]
```

3.18 Heap-Extract-Max

```
Heap-Extract-Max(A)
    if A.heapsize < 1
        Error "heap underflow"
    max <- A[1]
    A[1] <- A[A.heapsize]
    A.heapsize <- A.heapsize-1
    Max-Heapify(A,1)
    return max
```

3.19 Heap-Increase-Key

```
Heap-Increase-Key(A,i,key)
    if key < A[i]
        Error "The new key is less than the current value"
    A[i] <- key
    while i > 1 and A[Parent(i)] < A[i]
        Exchange A[i] with A[Parent(i)]
        i <- Parent(i)
```

3.20 Heap-Increase-Key

```
Heap-Increase-Key(A,i,key)
    if key < A[i]
        Error "The new key is less than the current value"
    A[i] <- key
    while i > 1 and A[Parent(i)] < A[i]
        Exchange A[i] with A[Parent(i)]
        i <- Parent(i)
```

3.21 Max-Heap-Insert

```
Max-Heap-Insert(A,key)
    A.heapsize <- A.heapsize+1
    A[A.heapsize] <- -∞
    Heap-Increase-Key(A,A.heapsize,key)
```

3.22 Stack-Empty

```
Stack-Empty(S)
    if S.top == 0
        return True
    return False
```

3.23 Push

```
Push(S,x)
    S.top <- S.top +1
    S[S.top] <- x
```

3.24 Pop

```
Pop(S)
    if Stack-Empty(S) == True
        Error "Underflow"
    else
        S.top <- S.top-1
        return S[S.top+1]
```

3.25 Enqueue

```
Enqueue(Q,x)
    Q[Q.tail] <- x
    if Q.tail == Q.length
        Q.tail <- 1
    else
        Q.tail <- Q.tail+1
```

3.26 Dequeue

```
Dequeue(Q)
    x <- Q[Q.head]
    if Q.head == Q.length
        Q.head <- 1
    else
        Q.head <- Q.head-1
    return x
```

3.27 List-Search

```
List-Search(L,x)
    x <- L.head
    while x ≠ NIL and x.key ≠ k
        x <- x.next
    return x
```

3.28 List-Insert

```
List-Insert(x)
    x.next <- L.head
    if L.head ≠ NIL
        L.head.prev <- x
    L.head <- x
    x.prev <- NIL
```

3.29 List-Delete (avec sentinelles)

```
List-Delete(L,x)
    if x.prev ≠ NIL
        x.prev.next <- x.next
    else
        L.head <- x.next
    if x.next ≠ NIL
        x.next.prev <- x.prev
```

3.30 List-Delete (sans sentinelles)

```
List-Delete(L,x)
    x.prev.next <- x.next
    x.next.prev <- x.prev
```

3.31 Quicksort-Iterative

```
Quicksort-Iterative(A,p,r)[2]
    Let S be a new empty stack
    Push(S,[p,r]
    while IsEmpty[3](S) == False
        [p,r] <- Pop(S)[4]
        if p < r
            q <- Partition(A,p,r)
            Push(S,[p,q-1])
            Push(S,[q+1,r])
```

4 Complexité temporelle

4.1 Ordre de complexité

On distingue deux types de complexité : la complexité *temporelle* et la complexité *spatiale*. La complexité temporelle d'un algorithme correspond à son temps d'exécution (*running-time*). Tandis que la complexité spatiale correspond à la quantité de mémoire utilisé à l'exécution. Dans ce cours, on ne s'intéressera qu'à la complexité temporelle.

On distingue ensuite trois types de complexité temporelle : la complexité dans le *pire cas*, dans le *meilleur cas* et en *moyenne*. La complexité en moyenne ne se calcul pas en faisant la somme de la complexité dans le meilleur cas et le pire cas, le tout divisé par deux. La complexité temporelle en moyenne correspond au temps d'exécution obtenu *par l'expérience* en testant la fonction avec des arguments hétérogènes. Ainsi, la complexité en moyenne est le temps d'exécution auquel on peut s'attendre en moyenne. On parle aussi simplement de complexité d'un algorithme. Dans ce cas, la complexité doit inclure le pire et le meilleur cas. Par exemple, si dans le pire cas, un algorithme est en $\Theta(n^2)$ et dans le meilleur cas en $\Theta(n)$, on peut dire que l'algorithme est en $\mathrm{O}(n^2)$ ou encore qu'il est en $\Omega(n)$. Attention, il n'y a jamais qu'une seule réponse valable pour la complexité ! Si un algorithme est en $\mathrm{O}(n^2)$, alors il est aussi en $\mathrm{O}(n^2 \ln(n))$ ou en $\mathrm{O}(n^3)$.
Il est bon d'avoir ces inégalités en tête :

$$1 << \ln(n) << n << n\ln(n) << n^2 << n^3 << n^4 << ... << 2^n << 3^n << ... << n!$$

Lorsque l'algorithme est en $\Theta(1)$, on dit qu'il a une complexité *constante*. Lorsqu'il est en $\Theta(\ln(n))$, on dit qu'il a une complexité *logarithmique*. Voici une liste de vocabulaire des cas les plus courants. Cette liste n'est pas à connaître mais il est bon de la maîtriser.

- $\Theta(1)$: constante
- $\Theta(\ln(n))$: logarithmique – $\Theta(\log_2(n)) = \Theta(\ln(n)) = \Theta(\log_b(n))$ pour tout $b > 1$
- $\Theta(n)$: linéaire
- $\Theta(n\ln(n))$: quasi-linéaire
- $\Theta(n^2)$: quadratique

[2] Sami, le majorant de la promo 2021-2022 des L1 MIDO — qui a également majoré cette matière —, avait fait une vidéo expliquant le fonctionnement de cet algorithme : https://youtu.be/0qha8OVn4kg

[3] Cette fonction est exactement la même que `Stack-Empty` (*cf.* 3.22). Le prof avait juste changé le nom.

[4] Pour rappel, dans cette fonction, S est un tableau de *tableaux à deux éléments*.

- $\Theta(n^3)$: cubique
- $\Theta(n^4)$: quartique
- $\Theta(2^n)$ et $\Theta(3^n)$: exponentielle
- $\Theta(n!)$: factorielle

Notez qu'il faut non seulement connaître la complexité temporelle des algorithmes de tri, mais aussi celle de tous les algorithmes vus dans le cours.

4.2 Algorithmes de tri

Voici un tableau représentant la complexité temporelle des algorithmes de tri.

	Meilleur des cas	Complexité moyenne	Pire des cas
Insertion-Sort	$\Theta(n)$	$\Theta(n^2)$	$\Theta(n^2)$
Quicksort	$\Theta(n\ln(n))$	$\Theta(n\ln(n))$	$\Theta(n^2)$
Merge-Sort	$\Theta(n\ln(n))$	$\Theta(n\ln(n))$	$\Theta(n\ln(n))$
Heapsort	$\Theta(n\ln(n))$	$\Theta(n\ln(n))$	$\Theta(n\ln(n))$

L'algorithme de tri le plus *chronophage* est `Insertion-Sort`, appelé *tri par insertion* en français. Cela n'est pas étonnant car il s'agit de l'algorithme de tri le plus intuitif. En effet, il s'agit de l'agorithme de tri que l'on suivrait naturellement pour trier un paquet de cartes. Si l'on a 32 cartes à trier, on étalerait les cartes dans l'une de nos mains. Appelons première carte la carte plus à gauche et dernière carte celle la plus à droite. On comparerait la deuxième carte avec la première, et on l'insèrerait avant ou après la première carte selon la comparaison. Puis, on comparerait la troisième carte avec les deux cartes triées précédemment et on l'insèrerait parmi les deux cartes triées, etc. `Insertion-Sort` est simplement un algorithme de tri introductif servant à faire la transition avec les autres algorithmes de tri qui sont moins naturels mais plus efficaces. Ainsi, il n'est pas étonnant que dans le meilleur des cas, `Insertion-Sort` soit le seul à avoir un temps d'exécution de $\Theta(n)$ et que dans le pire des cas, `Insertion-Sort` ait la complexité la plus élevée possible parmi les algorithmes de tri, c'est-à-dire $\Theta(n^2)$. Et pour ne pas aller dans le sens de `Insertion-Sort`, la complexité moyenne est la pire possible, c'est-à-dire $\Theta(n^2)$.

`Merge-Sort` et `Heapsort` sont en $\Theta(n\ln(n))$ dans tous les cas. Ces algorithmes sont donc moins risqués que `Quicksort` qui est en $\Theta(n^2)$ dans le pire des cas. En pratique, c'est cependant le tri rapide qui est le plus utilisé car son pire des cas est usuellement rare et qu'il est expérimentalement meilleur que `Merge-Sort` et `Heapsort` en moyenne.

4.3 Autres algorithmes

Voici une liste de temps d'exécution d'algorithmes. Cette liste n'est pas exhautive.

	Temps d'exécution
Merge	$\Theta(n)$
Parent	$\Theta(1)$
Left	$\Theta(1)$
Right	$\Theta(1)$
Max-Heapify	$\mathrm{O}(\ln(n))$
Build-Max-Heap	$\mathrm{O}(n)$
Heap-Select-Max	$\mathrm{O}(\ln(n))$
Heap-Increase-Key	$\mathrm{O}(\ln(n))$
Max-Heap-Insert	$\mathrm{O}(\ln(n))$
Push	$\Theta(1)$
Pop	$\Theta(1)$
Enqueue	$\Theta(1)$
Dequeue	$\Theta(1)$
List-Search	$\mathrm{O}(n)$
List-Insert	$\Theta(1)$
List-Delete	$\Theta(1)$