EXERCICES AVEC COURS INTÉGRÉ SUR L'OPTIMISATION DE LA COMPLEXITE TEMPORELLE PAR TABLE DE HASHAGE

S. Labopin

Table des matières

1 Mesurer les ressources consommées par l'exécution d'un algorithme pour estimer sa qualité

1.1 Comprendre ce que requiert l'exécution d'un algorithme pour comprendre ce qui fait sa qualité

L'exécution d'un algorithme par un ordinateur requiert de la **mémoire**, du **temps**, et du **travail** à faire faire par son processeur. Les meilleurs algorithmes sont ceux qui minimisent ces trois types de ressources.

Exercice 1 (Mesurer les ressources consommées par l'exécution d'un algorithme pour estimer sa qualité)
On rappelle que lorsqu'un processus est lancé par le système d'exploitation, il lui alloue notamment un espace dans la mémoire virtuelle et des plages de temps pendant lesquelles ce processus peut utiliser le processeur.

1) **MÉMOIRE**
   a) Pourquoi l'exécution d'un algorithme nécessite-t-elle parfois de la mémoire permanente en plus de la mémoire vive ?
   b) Expliquer pourquoi la quantité de mémoire utilisée par un processus influence son temps d'exécution.
   c) Avec quelle unité peut-on mesurer cette quantité de mémoire ?
2) **TEMPS**
   a) Avec quelle unité peut-on mesurer le temps d'exécution du processus ?
   b) Est-ce que le temps pendant lequel le processeur effectue les opérations demandées par le processus est égal au temps d'exécution de ce processus ? Pourquoi ?
3) **TRAVAIL**
   a) Comment peut-on mesurer le travail que le processus demande de faire faire au processeur ?
   b) Parmis toutes les mesures susmentionnées, laquelle(/lesquelles) ne dépend-elle (/dépendent-elles) ni du matériel informatique utilisé ni de la charge du système ?

1.2 Temps d'exécution d'un algorithme

1.2.1 Mesurer le temps que met un processus pour exécuter l'algorithme

Exemple - Exercice 2 (Déterminer le temps d'exécution d'un algorithme)

1) Télécharger le fichier random_numbers.csv récupérable via `https://caltuli.online`.
2) Écrire un programme *Python* qui récupère les valeurs stockées dans ce fichier dans une liste en utilisant le module *Panda*.
3) Écrire une fonction *Python* `traverseList` » qui parcourt la liste passée en paramètre.
   Indication : Le mot-clé `pass` permet de ne pas faire d'erreur de syntaxe même si l'on ne fait rien à chaque tour de boucle.
4) En utilisant la fonction `perf_counter()` du module `time` et la fonction `traverseList`, déterminer le temps de parcours de cette liste.
5) Via une boucle, déterminer 20 fois d'affilée le temps de parcours de cette liste.
6) Qu'en déduire ? Le temps d'exécution d'un algorithme est-il vraiment bien défini ?

⚠ Deux processus qui exécutent le même algorithme sur la même machine peuvent ne pas l'exécuter à la même vitesse.

Exemple - Exercice 3 (Déterminer le temps d'exécution d'un algorithme)

1) Écrire un programme *Python* pour déterminer si la liste comprend un nombre positif à un chiffre.
2) Mesurer son temps d'exécution.

Exemple - Exercice 4 (Déterminer le temps d'exécution d'un algorithme)

1) Écrire un programme *Python* pour déterminer s'il existe dans la liste trois nombres consécutifs (mais non nécessairement écrits consécutivement).
2) Mesurer son temps d'exécution.

1.2.2 Mesurer le temps que le processeur consacre au travail demandé par un processus qui exécute l'algorithme

Remarque - Exercice 5

1) On pourrait essayer de mesurer le temps effectif passé par le processeur à exécuter l'algorithme, sans compter les interruptions et la gestion des autres processus. Pourquoi cette mesure dépend-elle encore de la machine utilisée ?
2) Le processeur exécute un programme sous forme d'instructions, chacune prenant un certain nombre de cycles d'horloge.
   a) Expliquez pourquoi compter le nombre de cycles d'horloge consommés par un algorithme est une mesure plus universelle que le temps d'exécution en secondes.
   b) Cette mesure peut-elle aussi être affectée par les caractéristiques du matériel ? Pourquoi certains processeurs accomplissent-ils plus d'instructions par cycle que d'autres ?

1.3 La complexité temporelle, une mesure du travail qui ne dépend que de l'algorithme

1.3.1 Recherche d'une unité pour mesurer le travail demandé au processeur par un processus qui exécute l'algorithme

Remarque - Exercice 6 (recherche d'une grandeur étalon pour mesurer le travail demandé au processeur par un processus qui exécute l'algorithme)
On ne peut pas généraliser la mesure des cycles d'horloge à toutes les machines, alors quelle autre unité choisir pour comparer objectivement deux algorithmes ?

1.3.2 L'opération élémentaire, une unité de mesure du travail qui ne dépend que de l'algorithme

Définition (Opération élémentaire)
On appelle **opérations élémentaires** les suivantes :

- Un accès mémoire pour lire ou écrire la valeur d'une variable ou d'une case d'un tableau.
- Une opération arithmétique entre entiers ou réels telle que l'addition, la soustraction, la multiplication, la division ou le calcul du reste d'une division entière.
- Une comparaison entre deux entiers ou réels.
- Une opération logique telle que ET, OU, NON.
- Un transfert de contrôle (« informer le processeur de l'adresse de la prochaine instruction à effectuer quand elle n'est pas l'adresse qui suit immédiatement celle de l'instrcution précédente »).

Exercice 7 (Mesurer le travail demandé au processeur pour exécuter une instruction en comptant le nombre d'opérations élémentaires)
Combien d'opérations élémentaires représentent les instructions suivantes écrites en *Python* ?

1) `a=5`
2) `c=a+b`

Exercice 8 (Compter le nombre d'opérations élémentaires qui composent un algorithme)
De combien d'opérations élémentaires se compose l'algorithme suivant ?

```
x=1
if x==0:
    x=1
else:
  x=0
```

⚠ Depuis le début de ce cours, on fait un **abus de langage en écrivantt « algorithme » à la place de « code source »**.

Exercice 9 (Compter le nombre d'opérations élémentaires qui composent un algorithme)
En supposant que la taille de la liste (1 000 000) est déjà écrite dans la mémoire, démontrer que l'algorithme suivant se compose de 5 999 998 opérations élémentaires :

```
# la variable list est un tableau contenant les valeurs de random_numbers.csv
    for n in list:
        pass
```

Indication : Il faut d'abord écrire « l' » algorithme qui correspond à ce code source.

1.3.3 La complexité temporelle

1.3.3.1 La suite qui, à la taille de l'entrée de l'algorithme, associe le nombre d'opérations élémentaires à effectuer

Définition (La complexité temporelle d'un algorithme)
La **complexité temporelle** d'un algorithme est le nombre d'opérations élémentaires que représente son exécution en fonction de la « taille » de son entrée.

⚠ **Cette notion de « taille » de l'entrée dépend du contexte.**
Elle peut désigner par exemple le nombre d'octets qu'occupent l'entrée, le nombre d'éléments qui la composent si c'est une liste ou bien tout simplement sa valeur.

Notation (La complexité temporelle d'un algorithme)
On notera souvent $T(n)$ la complexité temporelle de l'algorithme concerné.
Autrement dit, $T(n)$ désignera souvent le nombre d'opérations élémentaires que demande l'algorithme concerné quand son entrée est de taille n.

Exercice 10 (Déterminer la complexité temporelle d'un algorithme)
Dans l'exercice 9, on a compté 5 999 998 opérations élémentaires pour l'algorithme suivant :

```
étape 1 : i <- 1
étape 2 : tester si la valeur de i est égale à 1 000 000, le nombre d'éléments de la liste
étape 3 : si le test réussit, c'est la fin de l'algorithme (sinon continuer à l'étape suivante)
étape 4 : i <- i+1
étape 5 : aller à l'étape 2
```

Cependant, déterminer sa complexité temporelle ne fait pas sens car cet algorithme n'a pas d'entrée.
On considère alors que la liste `list` et sa taille t son passées en entrée :

```
étape 1 : i <- 1
étape 2 : tester si la valeur de i est égale à t
étape 3 : si le test réussit, c'est la fin de l'algorithme (sinon continuer à l'étape suivante)
étape 4 : i <- i+1
étape 5 : aller à l'étape 2
```

Déterminer la complexité temporelle de cet algorithme.

⚠ Il y a tout de même encore une ambiguïté : Un algorithme peut ne pas demander le même travail pour deux entrées différentes, même si elles ont la même taille. Par exemple, dans l'exercice 3, l'exécution peut détecter un nombre positif à un chiffre dès le début de son parcours de la liste passée en entrée. Et, au contraire, si la liste passée en entrée ne comporte pas un tel nombre, l'algorithme demande un travail beaucoup plus important requerant le parcours de toute la liste.
En effet, en toute rigueur on devrait définir les trois types de complexité temporelle suivantes :

- $T_{\texttt{worst}}(n)$: La complexité temporelle dans le pire des cas.
- $T_{\texttt{avg}}(n)$: La complexité temporelle moyenne.
- $T_{\texttt{best}}(n)$: La complexité temporelle dans le meilleur des cas.

Dans ce cours, on se limite à $T(n) = T_{\texttt{worst}}(n)$, c'est-à-dire le nombre maximal d'opérations élémentaires que demande l'algorithme pour une entrée de taille n.

Exercice 11 (Déterminer la complexité temporelle d'un algorithme)
Déterminer la complexité temporelle $T(n)$ pour chacun des algorithmes suivants :
⚠ **On négligera le calcul de la taille de la liste passée en entrée.**

1) La fonction définie dans l'exercice 3 qui détermine si la liste en entrée comprend un nombre positif à un chiffre.
2) La fonction définie dans l'exercice 4 qui détermine si la liste en entrée comprend trois nombres consécutifs (mais non nécessairement écrits consécutivement).
   Indication : On considère que le traitement de la liste $(1,1,...,2)$ est un des pires cas.

Exercice 12 (Déterminer la complexité temporelle d'une fonction récursive)

1) *a*) Sans chercher à optimiser, écrire une simple fonction itérative *Python* `expoSept` qui prend en entrée un nombre entier naturel n et qui renvoie en sortie 7^n.
   b) Mesurer son temps d'exécution pour différentes pour les entrées 0, 1, 2,..., 100 et tracer le nuage de points correspondant montrant l'évolution du temps d'exécution en fonction de la taille de l'entrée.
   c) Déterminer la complexité temporelle de `expoSept`.
2) *a*) En cherchant maintenant à optimiser, écrire une fonction récursive *Python* `expoSeptRapide` qui suit les mêmes spécifications que `expoSept` mais avec une complexité temporelle bien inférieure en remarquant que :

$$\forall\ p \in \mathbb{N},\ \begin{cases} 7^{2p} = 7^p \cdot 7^p \\ 7^{2p+1} = 7^p \cdot 7^p \cdot 7 \end{cases}$$

   b) Mesurer son temps d'exécution pour différentes pour les entrées 0, 1, 2,..., 100 et tracer le nuage de points correspondant montrant l'évolution du temps d'exécution en fonction de la taille de l'entrée.
   c) On note T la complexité temporelle de `expoSeptRapide`.
   Soit $n \in \mathbb{N}$.
      i) Exprimer $T(2n)$ et $T(2n+1)$ en fonction de $T(n)$.
      ii) En déduire que a et b tel que :

$$\forall\ u \in \mathbb{N},\ T\left(2^u\right) = au + b$$

      iii) On rappelle que : $\log_2(n) = \frac{\ln(n)}{\ln(2)}$.
      Démontrer que :

$$n = 2^u \iff u = \log_2(n)$$

      iv) En remarquant que T est une fonction croissante, démontrer que :

$$\forall\ n \in \mathbb{N},\ T(n) \leq a\log_2(n) + (a+b)$$

1.3.3.2 Comportements asymptotiques d'une suite numérique

Définition (Notations de Landau).
Soient $(u_n)_{n\in\mathbb{N}}$ et $(v_n)_{n\in\mathbb{N}}$ deux suites numériques.

- On dit que $\boldsymbol{u_n}$ **est négligeable devant** $\boldsymbol{v_n}$ et on note $\boxed{\boldsymbol{u_n = o(v_n)}}$ lorsque :

$$\frac{u_n}{v_n} \underset{n\to+\infty}{\to} 0$$

- On dit que $\boldsymbol{u_n}$ **est dominée par** $\boldsymbol{v_n}$ et on note $\boxed{\boldsymbol{u_n = O(v_n)}}$ lorsque :

$$\left(\frac{u_n}{v_n}\right)_{n\in\mathbb{N}} \text{ est une suite bornée}$$

- Le point précédent peut s'écrire autrement comme suit :

$$u_n = O(v_n) \iff \text{Il existe un réel } M \text{ tel que pour tout } n \in \mathbb{N},\ |u_n| \leq M \cdot |v_n|$$

Exemples (Comportement asymptotique de la complexité temporelle)

- $2n = O(n)$ car la suite $\left(\frac{2n}{n}\right)_{n\in\mathbb{N}} = (2)_{n\in\mathbb{N}}$ est une suite bornée (Elle est même constante.).
- $\ln(n) = o(n)$ car $\frac{\ln(n)}{n} \underset{n\to+\infty}{\to} 0$
- Une suite qui tend vers 0 étant en particulier bornée, on déduit du point précédent que $\ln(n) = O(n)$.
- $2n\ln(n) - 3n + 47 = O(n\ln(n))$

Abus de langage (« = » au lieu de « ∈ »)

- Quand on écrit par exemple « $3n^2+4n = O\left(n^2\right)$ », il ne s'agit pas d'une égalité entre deux suites numériques mais plutôt de l'appartenance de la suite $\left(3n^2+4n\right)_{n\in\mathbb{N}}$ à l'ensemble de toutes les suites qui sont dominées par la suite $\left(n^2\right)_{n\in\mathbb{N}}$.
- La notation « $O\left(u_n\right)$ » désigne en toute rigueur **l'ensemble de toutes les suites qui sont** dominées par la suite $\left(u_n\right)_{n\in\mathbb{N}}$.
  Cependant, lorque l'on écrit « $O\left(u_n\right)$ », on sous-entend en fait la plupart du temps **une suite qui est** dominée par la suite $\left(u_n\right)_{n\in\mathbb{N}}$.
- C'est pourquoi, quand on s'intéresse au comportement asymptotique des suites comme on va d'ailleurs le faire avec le nombre d'opérations élémentaires d'un algorithme en fonction de la taille de son entrée, c'est du pareil au même de considérer la suite $\left(3n^2+4n\right)_{n\in\mathbb{N}}$ ou de considérer la suite $\left(7n^2+\ln(n)+17\right)_{n\in\mathbb{N}}$ car ce sont toutes les deux des suites dominées par la suite $\left(n^2\right)_{n\in\mathbb{N}}$ (et non négligeable devant elle).
  On dit simplement que l'on considère une suite en $O\left(n^2\right)$.

1.3.3.3 Le comportement asymptotique de la complexité temporelle, une mesure encore bien plus objective de la qualité d'un algorithme

Exercice 13 (Déterminer le comportement asymptotique de la complexité temporelle d'un algorithme)
Utiliser les notations de Landau pour écrire le plus simplement possible le comportement asymptotique de la complexité temporelle pour chacun des algorithmes suivants :

⚠ **On remarquera au passage que la non prise en compte des écritures des entrées et des sorties dans la mémoire en tant qu'opération opération élémentaire n'affecte pas la complexité temporelle de ces trois algorithmes.**

1) La fonction *Python* `traverseList` définit dans l'exercice 2.
2) La fonction définie dans l'exercice 3 qui détermine si la liste en entrée comprend un nombre positif à un chiffre.
3) La fonction définie dans l'exercice 4 qui détermine si la liste en entrée comprend trois nombres consécutifs (mais non nécessairement écrits consécutivement).

Exercice 14 (Déterminer le comportement asymptotique de la complexité temporelle d'un algorithme)
Déterminer le comportement asymptotique de la complexité temporelle des fonctions `expoSept` et `expoSeptRapide` définies dans l'exercice 12.

Remarque - Exercice 15
En s'inspirant de l'exercice 14, expliquer pourquoi la complexité temporelle de la lecture des paramètres ou de l'écriture de la sortie n'est pas toujours $O(1)$.

1.4 Complexité temporelle de quelques algorithmes de tri usuels

1.4.1 Tri par insertion

Exercice 16 (Complexité temporelle du tri par insertion)

1) Écrire une fonction *Python* `tri_par_insertion` qui trie sur place la liste passée en paramètre dans l'ordre croissant en implémentant cette fonction suivant l'algorithme du tri par insertion.
   Indication : Le tri par insertion consiste à déplacer un par un les éléments de la liste à partir du deuxième vers sa position correcte en l'insérant parmis les éléments déjà triés à sa gauche.
2) Mesurer le temps d'exécution des appels « `tri_par_insertion(l)` » avec `l` la liste des n premiers nombres du fichiers random_numbers.csv pour n allant de 1 à 1 000.
3) Tracer le nuage de points correspondant montrant l'évolution du temps d'exécution du tri par insertion en fonction de la longueur de la liste à trier.
4) Déterminer la complexité temporelle du tri par insertion (ou plutôt son comportement asymptotique).

1.4.2 Tri par fusion

Exercice 17 (Complexité temporelle du tri par fusion)
Le tri par fusion est l'algorithme récursif suivant :

```
Si la liste n'a qu'un élément, elle est déjà triée
Sinon, la diviser en deux listes de taille à peu près égale.
Trier ces deux sous-listes avec l'algorithme du tri par fusion.
Fusionner les deux listes triées obtenues en une seule liste.
```

1) Écrire une fonction *Python* `tri_par_fusion` qui prend une liste de nombres en entrée et qui renvoie en sortie une liste contenant les mêmes nombres dans l'ordre croissant en implémentant cette fonction suivant l'algorithme du tri par fusion.
2) Mesurer le temps d'exécution des appels « `tri_par_fusion(l)` » avec `l` la liste des n premiers nombres du fichiers random_numbers.csv pour n allant de 1 à 1 000.
3) Tracer le nuage de points correspondant montrant l'évolution du temps d'exécution du tri par fusion en fonction de la longueur de la liste à trier.
   Le faire apparaitre avec celui du tri par insertion implémenté dans l'exercice 16 sur le même graphique afin de pouvoir bien comparer expérimentalement ces deux algorithmes.
4) Cette question a pour but de déterminer le comportement asymptotique de la complexité temporelle du tri par fusion.
   On note $(T(n))_{n\in\mathbb{N}^*}$ la complexité temporelle du tri par fusion.
   a) Démontrer que :
   $$T\left(2^n\right) = O\left(2^n\right) + 2T\left(2^{n-1}\right)$$
   b) En déduire que :
   $$T\left(2^n\right) = O\left(n2^n\right)$$
   c) Soit $n \in \mathbb{N}^*$.
   Déterminer (en fonction de n) le petit entier x tel que :
   $$n \leq 2^x$$
   Indication : Utiliser la fonction logarithme népérien et la fonction partie entière.
   d) En déduire que :
   $$\boxed{T(n) = O\left(n\log_2(n)\right)}$$
   Indications :
   - Comprendre d'abord pourquoi la suite $(T(n))_{n\in\mathbb{N}^*}$ est croissante.
   - $\log_2(n) = \frac{\ln(n)}{\ln(2)}$

Exercice 18 (Quand le tri par insertion est plus rapide que le tri par fusion)
Sur une certaine machine et relativement à un certain modèle de calcul, le tri par insertion demande $8n^2$ opérations élémentaires et le tri par fusion demande $64n\ln(n)$ opérations élémentaires.
Déterminer les valeurs de n telles que le tri par insertion demande moins d'opérations élémentaires que le tri par fusion.

1.4.3 Tri à bulles

Exercice 19 (Complexité temporelle du tri à bulles)
Le tri à bulles est l'algorithme suivant :

```
Entrée : une liste l
Sortie : une liste croissante sorted_list dont les éléments sont ceux de l
sorted_list <- l
pour i allant de 0 à taille(l)-1 :
  faire pour j allant de taille(l)-1 à i+1:
    faire si sorted_list[j]<sorted_list[j-1]:
       alors permuter sorted_list[j-1] et sorted_list[j]
```

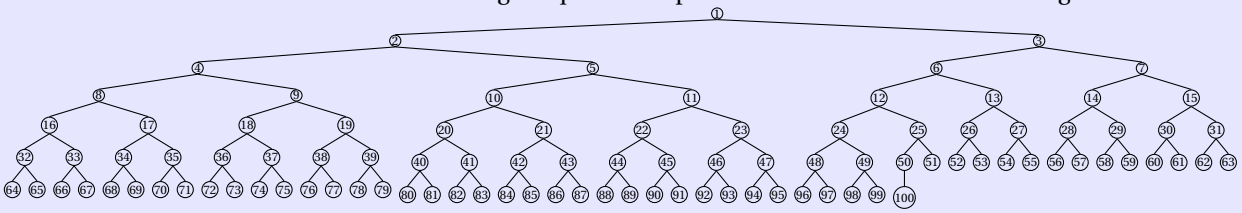
1) Écrire une fonction *Python* `tri_a_bulles` qui prend une liste de nombres en entrée et qui renvoie en sortie une liste contenant les mêmes nombres dans l'ordre croissant en implémentant cette fonction suivant l'algorithme du tri à bulles.
2) Mesurer le temps d'exécution des appels « `tri_a_bulles(l)` » avec `l` la liste des n premiers nombres du fichiers random_numbers.csv pour n allant de 1 à 1000.
3) Tracer le nuage de points correspondant montrant l'évolution du temps d'exécution du tri à bulles en fonction de la longueur de la liste à trier.
   Le faire apparaitre avec ceux des autres tris implémentés dans les exercices précédents sur le même graphique afin de pouvoir bien comparer expérimentalement tous ces algorithmes.
4) Déterminer le comportement asymptotique de la complexité temporelle du tri à bulles

1.4.4 Tri par tas

Remarque (Structurer dans un tableau l'information d'un arbre binaire quasi-complet)
On peut écrire les 100 premiers nombres entiers naturels comme suit, avec 2^0 nombre à l'étage 0, 2^1 nombres à l'étage 1, 2^2 nombres à l'étage 2, 2^3 nombres à l'étage 3, etc... jusqu'à n'avoir que le dernier étage qui n'est éventuellement pas rempli :

```
étage 0 : 1
étage 1 : 2 3
étage 2 : 4 5 6 7
étage 3 : 8 9 10 11 12 13 14 15
étage 4 : 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31
étage 5 : 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52 53 54 55 56 57 58 59 60 61 62 63
étage 6 : 64 65 66 67 68 69 70 71 72 73 74 75 76 77 78 79 80 81 82 83 84 85 86 87 88 89 90 91 92 93 94 95 96 97 98 99 100
```

Comme chaque étage comporte deux fois plus d'éléments que le précédent à l'exception du dernier qui en comporte encore moins, on peut étiqueter les nœuds d'un arbre binaire avec ces 100 indices suivant les mêmes étages qui correspondent alors aux différentes générations :



On comprend alors naturellement comment structurer l'information d'un arbre binaire avec un tableau : On écrit dans le tableau les valeurs stockées dans les nœuds de l'arbre en lisant de gauche à droite puis de haut en bas.
Et on peut retrouver l'information de l'arbre à partir d'un tel tableau :

- On peut d'abord retrouver les générations. En effet, la génération i stocke les valeurs $e_{2^i}, ..., e_{2^{i+1}-1}$ ou les premières de ces valeurs si la génération i est la dernière génération :

$$\Big[\underbrace{e_1}_{\text{gén. 0}}, \underbrace{e_2, e_3}_{\text{gén. 1}}, \underbrace{e_4, e_5, e_6, e_7}_{\text{gén. 2}}, \underbrace{e_8, \dots, e_{15}}_{\text{gén. 3}}, \underbrace{e_{16}, \dots, e_{31}}_{\text{gén. 4}}, \underbrace{e_{32}, \dots, e_{63}}_{\text{gén. 5}}, \underbrace{e_{64}, \dots, e_{100}}_{\text{gén. 6}}\Big]$$

- Mais on voit surtout immédiatement en regardant l'arbre ci-desssus que l'enfant gauche et l'enfant droit du nœud d'indice p sont respectivement les nœuds d'indices $2p$ et $2p+1$.
  On en déduit alors aussi immédiatement que le parent d'un nœud n est $\lfloor \frac{n}{2} \rfloor$:

$$\boxed{\text{gauche}(i) = 2i} \qquad \boxed{\text{droite}(i) = 2i+1} \qquad \boxed{\text{parent}(i) = \lfloor \frac{i}{2} \rfloor}$$

Remarque (Tableaux et listes en *Python*)
La structure de donnée rigide « tableau » n'est pas implémentée par *Python*.
C'est pourquoi on utilisera une liste plutôt qu'un tableau.

Exercice 21 (Complexité temporelle de la récupération d'une valeur dans un arbre binaire)
Un arbre binaire quasi-complet est un arbre du type précédent, c'est-à-dire un arbre binaire dont toutes les générations sont complètes à l'exception de la dernière.

1) Combien de nœuds peut-il y avoir dans la $i^{\text{ème}}$ génération d'un arbre binaire quasi-complet ?
2) *a*) Combien y a-t-il de générations dans un arbre binaire quasi-complet comportant n nœuds ?
   b) En déduire le comportement asymptotique de la complexité temporelle de la récupération d'une valeur dans un arbre binaire.

Définition (tas et tas max)

- On appelle **tas** un tableau quand on le voit comme l'arbre binaire quasi-complet.
- On appelle **tas max** un tas dont la valeur de chaque nœud est supérieure ou égale à la valeur de ses deux nœuds enfants.

Exercice 22 (Opération sur les tas et **tri par tas**)

1) Écrire une fonction récursive *Python* `entasser_max` d'interface suivante :
   ⚠ Quand on parle de tas, on numérote à partir de 1 et non à partir de 0.
   ⚠ Dans la spécification ci-dessous, le mot « taille » ne désigne pas la longueur de la liste (le nombre d'éléments).

entasser_max(l,taille,indice)	

Interface	entasser_max(l,taille,indice)
	Entrée :
	  l : liste
	  taille : entier
	  indice : entier
	Préconditions :
	  0 ≤ taille ≤ longueur(l)
	  0 ≤ indice - 1 ≤ taille -1
	  $[l[0], l[1], ..., l[taille-1]]$ est un tas
	  dans ce tas, les arbres de racines gauche(indice) et doite(indice) sont des tas max
	Sortie :
	  aucune ; la liste est modifiée sur place
	Postcondition :
	  dans le tas $[l[0], l[1], ..., l[taille-1]]$, l'arbre de racine d'indice i est un tas max

2) En utilisant la fonction `entasser_max`, écrire une fonction *Python* `construire_tas_max` d'interface suivante :
   Indication : Le dernier nœud qui n'est pas une feuille est la parent du dernier nœud.

construire_tas_max(l)	

Interface	construire_tas_max(l)
	Entrée :
	  l : liste
	Précondition :
	  aucune
	Sortie :
	  aucune ; la liste est modifiée sur place
	Postcondition :
	  l est un tas max

3) En utilisant les fonctions `construire_tas_max` et `entasser_max`, écrire une fonction *Python* `tri_par_tas` d'interface suivante :
   Indication :
   - trasformer la liste passée `l` en un tas max; le premier élément de `l` est alors le plus grand
   - permuter le premier élément de `l` avec le dernier de façon à ce que le plus grand soit maintenant le dernier et que l'entrée (l,len(l)-1,1) satisfasse la précondition de la spécification de `entasser_max`
   - puis recommencer pour avoir le deuxième plus grand en avant dernière position
   - et ainsi de suite jusqu'à appeler `entasser_max` avec une taille de 2

tri_par_tas(l)	

Interface	tri_par_tas(l)
	Entrée :
	  l : liste
	Précondition :
	  aucune
	Sortie :
	  aucune ; la liste est modifiée sur place
	Postcondition :
	  l contient les mêmes valeurs mais dans l'ordre croissant

Exercice 23 (Complexité temporelle du tri par tas)

1) Mesurer le temps d'exécution des appels « `tri_par_tas(l)` » avec `l` la liste des n premiers nombres du fichiers random_numbers.csv pour n allant de 1 à 1 000.
2) Tracer le nuage de points correspondant montrant l'évolution du temps d'exécution du tri par tas en fonction de la longueur de la liste à trier.
   Le faire apparaitre avec ceux des autres tris implémentés dans les exercices précédents sur le même graphique afin de pouvoir bien comparer expérimentalement tous ces algorithmes.
3) Déterminer le comportement asymptotique de la complexité temporelle du tri par tas.

1.4.5 Tri rapide

Exercice 24 (Complexité temporelle du tri rapide)
Le tri rapide est décrit dans l'article de *Wikidépia* « Tri rapide » :

> *« La méthode 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, 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. »*

Le livre « Introduction à l'algorithmique » de T. Cormen, C. Leiserson, R. Rivest et C. Stein propose l'algorithme suivant pour le partitionnement des éléments d'un tableau `A` de l'indice `p` à l'indice `r` avec pour pivot la valeur d'indice `r` :

```
partition(A,p,r)
    x ← A[r]
    i ← p-1
    pour j de p à r-1
      si A[j] ≤ x
          i ← i+1
          permuter A[i] et A[j]
    permuter A[i+1] et A[r]
    retourner i+1
```

1) Adapter l'algorithme précédant afin d'écrire une fonction *Python* `partition(l,p,r,pivot_indice)` qui prend un quatrième argument permettant de choisir l'indice du pivot et de ne pas nécessairement s'appuyer sur le dernier indice.
2) Écrire une fonction *Python* `tri_rapide` implémentant l'algorithme du tri rapide et prenant pour pivot l'indice de la médiane des valeurs extrêmes et de la valeur centrale de la plage à patitionner.
3) Mesurer le temps d'exécution des appels « `tri_rapide(l)` » avec `l` la liste des n premiers nombres du fichiers random_numbers.csv pour n allant de 1 à 1 000.
4) Tracer le nuage de points correspondant montrant l'évolution du temps d'exécution du tri rapide en fonction de la longueur de la liste à trier.
   Le faire apparaitre avec ceux des autres tris implémentés dans les exercices précédents sur le même graphique afin de pouvoir bien comparer expérimentalement tous ces algorithmes.
5) a) Déterminer le comportement asymptotique de la complexité temporelle du tri rapide.
   b) Pourquoi ce comportement asymptotique ne correspond pas à ce qu'on observe sur ce graphique ?

Remarque
La complexité temporelle **dans le pire des cas** du tri rapide est $O\left(n^2\right)$ mais sa complexité temporelle **moyenne** est $O(n\ln(n))$.

1.5 Complexité spatiale

Définition (La complexité spatiale d'un algorithme)
La **complexité spatiale** d'un algorithme est la quantité de mémoire nécessaire à son exécution en fonction de la taille de son entrée.

Exercice 25 (Déterminer la complexité spatiale d'un algorithme)
Déterminer la complexité spatiale pour chacun des algorithmes suivants :

1) La fonction *Python* `traverseList` définit dans l'exercice 2.
2) La fonction définie dans l'exercice 3 qui détermine si la liste en entrée comprend un nombre positif à un chiffre.
3) La fonction définie dans l'exercice 4 qui détermine si la liste en entrée comprend trois nombres consécutifs (mais non nécessairement écrits consécutivement).
4) Les fonctions `expoSept` et `expoSeptRapide` définies dans l'exercice 12.
5) Les fonctions implémentées dans la section précédente.

2 Table de hashage et diminution de la complexité temporelle

2.1 Une première idée : Une structure de donnée très grande mais qui se fouille rapidement

Exercice 26 (Une structure de donnée très grande mais qui se fouille rapidement)

1) Écrire une fonction *Python* `max_longueur_consecutifs` d'interface suivante :

max_longueur_consecutifs(l)	

Interface	`max_longueur_consecutifs(l)`
	`Entrée :`
	`  l : liste`
	`Préconditions :`
	`  Pour tout élément e de la liste l, -1 000 000 < e < 1 000 000`
	`Sortie :`
	`  diametre : entier`
	`Postcondition :`
	`  diametre est le nombre maximum de nombres positifs consécutifs qui sont des éléments de l`

2) Mesurer le temps d'exécution des appels « `max_longueur_consecutifs(l)` » avec `l` la liste des n premiers nombres du fichiers random_numbers.csv pour n parcourant `list(range(0,100_000+1_000,5_000))`.
3) Écrire une fonction *Python* `construire_pseudo_table_de_hashage` d'interface suivante :

construire_pseudo_table_de_hashage(l)	

Interface	`construire_pseudo_table_de_hashage(l)`
	`Entrée :`
	`  l : liste`
	`Préconditions :`
	`  Pour tout élément e de la liste l, -1 000 000 < e < 1 000 000`
	`Sortie :`
	`  pth : liste à 1 000 000 d'éléments`
	`Postcondition :`
	`  pout tout i de 0 à 999 999, l'élément d'indice i vaut True si l'entier i est un élément de l ou bien vaut False sinon.`

4) En déduire une fonction *Python* `max_longueur_consecutifs_rapide` de même interface que `max_longueur_consecutifs` mais beaucoup plus rapide en utilisant la fonction `construire_pseudo_table_de_hashage`.
5) Mesurer le temps d'exécution des appels « `max_longueur_consecutifs_rapide(l)` » avec `l` la liste des n premiers nombres du fichiers random_numbers.csv pour n parcourant `list(range(0,100_000+1_000,5_000))`.
6) Tracer les nuages de points correspondant montrant l'évolution du temps d'exécution de chacun de ces deux algorithmes en fonctions de la taille de la liste passée en entrée.
7) Déterminer le comportement asymptotique de la complexité temporelle et de la complexité spatiale des foncions `max_longueur_consecutifs` et `max_longueur_consecutifs_rapide`.

2.2 Définition et exemple d'une table de hashage

Définition (table de hashage)
Une **table de hashage** est une structure de donnée qui stocke des couples « `(clef,valeur)` » de façon à ce que la récupération d'une valeur à partir de sa clef soit très rapide.
Plus précisément :

- Les opérations de cette structure de données sont les suivantes :
  - l'insertion d'un couple « `(clef,valeur)` » si la table de hashage ne contient pas actuellement d'élément ayant la même clé (⚠ Une même valeur peut être stockées à deux endroits différents avec des clés différentes mais une clé ne peut correspondre qu'à une unique valeur stockée.) ;
  - la récupération d'une valeur à partir de sa clef ;
  - la suppression d'un couple « `(clef,valeur)` » ;
  - le parcours de l'ensemble des couples « `(clef,valeur)` » qui sont actuellement stockés dans la table de hashage.
- Une **table de hashage** est un tableau de listes.
- Les éléments de ces listes sont les couples « `(clef,valeur)` » qui sont actuellement stockés dans cette table de hashage.
- Cette structure de données « table de hashage », en plus de comprendre ce tableau de listes et les opérations susmentionnées, comprend une fonction appelée « **fonction de hashage** ». Cette fonction prend un paramètre `c` et renvoie l'indice de l'unique liste dans laquelle on peut insérer un couple « `(clef,valeur)` » dont la clé `clef` est égale à `c`.

Définition (collision dans une table de hashage)
⚠ Deux clés différentes peuvent être associées à deux indices différents par la fonction de hashage. C'est pourquoi on appelle **collision** l'insertion de deux couples « `(clef,valeur)` » différents au même indice de la table. Et c'est aussi pourquoi on a conçu un tableau de listes de couples « `(clef,valeur)` » plutôt qu'un tableau de couples « `(clef,valeur)` » ; l'insertion d'un nouvel élément à un indice déjà « occupé » se fait en complétant cette liste.

Exemple (implémentation d'une table de hashage en *C*)

```c
# define HashTableSize 65011
# define MaxVariableNameLength 256
typedef struct cons {char *key; double value; struct cons * cdr;} cons, *list;
list variables[HashTableSize] = {0};
typedef struct {char* key; int index;} variable_location;
unsigned long int hash(char * s) {
  unsigned long int hash = 5381 ;
  while (* s) {
    int c = * s ;
    hash = ((hash << 5) + hash) + c ;
    s++ ;
  }
  return hash ;
}
int get_hash_table_index(char * s) {
  return hash(s) % HashTableSize;
}
variable_location create_variable(char * variable_name) {
  variable_location vl;
  vl.key = variable_name;
  vl.index = get_hash_table_index(variable_name);
  list localized_value = malloc(sizeof(cons));
  localized_value->key = vl.key;
  localized_value->value = 0;
  localized_value->cdr = variables[vl.index];
  variables[vl.index] = localized_value;
  return vl;
}
```

Exercice 27 (Implémentation en *C* d'une table de hashage)

1) Expliquer l'implémentation en *C* de la table de hashage de l'exemple précédent pour stocker les variables et leur valeur.
2) Implémenter la fonction de prototype « `double * get_variable_address(char * name)` » qui prend une chaîne de caractères en paramètre et qui renvoie un pointeur pointant sur l'adresse où est écrite la valeur de la variable qui a pour indentifiant cette chaîne de caractères.

2.3 Implémentation en *Python* d'une table de hashage et application

Exemple (Implémentation en *Python* d'un dictionnaire associatif par une table de hashage)
En *Python*, un dictionnaire associatif est implémenté avec une table de hashage.
La syntaxe suivante initialise un dictionnaire associatif `th` ne contenant aucun couple « `(clef,valeur)` » :

```
th=dict()
```

La syntaxe suivante insère dans la table de hashage `th` le couple « `("table de hashage","hash table")` » :

```
th["table de hashage"]="hash table"
```

La syntaxe suivante récupère la valeur associée à la clé `"table de hashage"` par la table de hashage `th` :

```
th["table de hashage"]
```

Exercice 28

1) Implémenter une fonction *Python* `max_longueur_consecutifs_rapide_avec_hashage` de même interface que `max_longueur_consecutifs` mais beaucoup plus rapide en utilisant une table de hashage (par le biais d'un dictionnaire associatif).
2) Mesurer le temps d'exécution des appels « `max_longueur_consecutifs_rapide_avec_hashage(l)` » avec `l` la liste des n premiers nombres du fichiers random_numbers.csv pour n parcourant `list(range(0,100_000+1_000,5_000))`.
3) Tracer sur le même graphique les nuages de points correspondant montrant l'évolution du temps d'exécution de chacun des algorithmes `max_longueur_consecutifs_rapide_avec_hashage` et `max_longueur_consecutifs_rapide` en fonctions de la taille de la liste passée en entrée.