TD - Preuves d'algorithmes
Terminaison - Complexité
Corrigé

Exercice 1 (Échauffement - Notations de Landau).
Simplifier les écritures suivantes :

1. $O(n+1)$
2. $O(3n+3)$
3. $O(\frac{n(n+1)}{2})$
4. $O(n^2e^{42} + 3 \times 2^{3n-1})$
5. $O(\log_2(n+1) + \ln(2n^2))$

Corrigé de l'exercice 1. *[Retour à l'énoncé]*

1. $O(n+1) \in O(n)$
2. $O(3n+3) \in O(n)$
3. $O(\frac{n(n+1)}{2}) \in O(n^2)$
4. $O(n^2e^{42} + 3 \times 2^{3n-1}) \in O(8^n)$
5. $O(\log_2(n+1) + \ln(2n^2)) \in O(\log_2(n))$

Exercice 2 (Échauffement - Complexité des boucles).
Dans toutes les questions, n et $m < \frac{n}{2}$ désignent des entiers naturels. i, j, k sont des entiers préalablement déclarés. x est un entier préalablement défini. En détaillant vos calculs, déterminer la complexité temporelle de chacun des codes suivants.

```
// code 1
for (i = 0; i < n; i++)
  for (j = 0; j < n; j++)
    x = x + 1;

// code 2
for (i = 0; i < n; i++)
  for (j = 0; j < i; j++)
    x = x + 1;

// code 3
for (i = m; i < n-m; i++) {
  for (j = i-m; j < i+m; j++)
    x = x + 1;

// code 4
for (i = 0; i < n; i++)
  for (j = 0; j < i; j++)
    for (k = 0; k < j; k++)
      x = x + 1;
```

```
// code 5
int i = n;
while (i > 1) {
  x = x + 1;
  i = i / 2;
}

// code 6
i = n;
while (i > 1) {
  for (j = 0; j < n; j++)
    x = x + 1;
  i = i / 2;
}

# code 7
i = n;
while (i > 1) {
  for (j = 0; j < i; j++)
    x = x + 1;
  i = i / 2;
}
```

Corrigé de l'exercice 2. *[Retour à l'énoncé]*

Notons c_+ et $c_=$ les coûts respectivement associés à une addition et une affectation, et $\alpha = c_+ + c_=$.

— Le coût de `code 1` peut être estimé par :

$$C(n) = \sum_{i=0}^{n-1} \sum_{j=0}^{n-1} \alpha = \alpha n^2$$

D'où une complexité asymptotique en $O(n^2)$.

— Pour `code 2`, la borne supérieure sur j est modifiée.

$$C(n) = \sum_{i=0}^{n-1}\sum_{j=0}^{i-1}\alpha = \sum_{i=0}^{n-1}\alpha i = \alpha\frac{n(n-1)}{2}$$

D'où une complexité en $O(n^2)$.

— Pour `code 3`, on a :

$$C(n) = \sum_{i=m}^{n-m-1}\sum_{j=i-m}^{i+m-1}\alpha = \sum_{i=m}^{n-m-1}\alpha\left(i+m-1-(i-m)+1\right) = \sum_{i=m}^{n-m-1}\alpha\left(2m\right) = 2\alpha m(n-2m)$$

D'où une complexité en $O(n)$.

— Pour `code 4`, on a :

$$\begin{aligned}
C(n) &= \sum_{i=0}^{n-1}\sum_{j=0}^{i-1}\sum_{k=0}^{j-1}\alpha = \alpha\sum_{i=0}^{n-1}\sum_{j=0}^{i-1}j = \alpha\sum_{i=0}^{n-1}\frac{i(i-1)}{2} \\
&= \frac{\alpha}{2}\left(\sum_{i=0}^{n-1}i^2 - \sum_{i=0}^{n-1}i\right) = \frac{\alpha}{2}\left(\frac{(n-1)n(2(n-1)+1)}{6} - \frac{(n-1)n}{2}\right) = \frac{\alpha}{2}\left(\frac{(n-1)n(2n-1)}{6} - \frac{3(n-1)n}{6}\right) \\
&= \frac{\alpha(n-1)n}{2}\left(\frac{(2n-1)}{6} - \frac{3}{6}\right) = \frac{\alpha(n-1)n}{2}\frac{(2n-4)}{6} = \frac{\alpha(n-1)n}{2}\frac{(n-2)}{3} \\
&= \alpha\frac{n(n-1)(n-2)}{6}
\end{aligned}$$

D'où une complexité en $O(n^3)$.

Pour tout entier positif n non nul, on rappelle l'existence d'un unique entier positif p tel que $2^p \leq n < 2^{p+1}$. On montre que $p = \lfloor \log_2(n) \rfloor$. Chacune des boucles des trois codes `5`, `6`, `7` termine en exactement p tours, la valeur finale contenue dans la variable `i` étant `1`.

— En notant $c_/$ le coût d'une division, et $\beta = c_+ + c_/ + 2c_=$, le coût de `code 5` est donné par :

$$C(n) = \sum_{k=1}^{p}\beta = \beta p$$

D'où une complexité en $O(\log_2(n))$.

— Pour `code 6`, on remarque la présence de la boucle `for` qui itère n fois. Notons $\beta' = c_+ + c_=$ et $\gamma' = c_/ + c_=$. Le coût de cet algorithme est évalué par :

$$C(n) = \sum_{k=1}^{p}\left[\gamma' + \sum_{j=0}^{n-1}\beta'\right] = \sum_{k=1}^{p}(\gamma' + \beta' n) = (\gamma' + \beta' n)p$$

D'où une complexité en $O(n\log_2(n))$.

— Pour `code 7`, le calcul est plus délicat en raison de la dépendance en i dans la boucle `for`. Le coût est de la forme :

$$C(n) = \gamma' p + \beta' \times \left(n + \left\lfloor\frac{n}{2}\right\rfloor + \left\lfloor\frac{n}{4}\right\rfloor + \cdots + 2\right)$$

Pour évaluer la complexité asymptotique de cet algorithme, on peut se contenter de majorer la somme entre parenthèses [1]. Puisqu'il existe un entier p tel que $2^p \leqslant n < 2^{p+1}$, pour tout entier k compris entre 0 et $(p-1)$, on a :

$$\frac{n}{2^k} < 2^{p-k+1}$$

Le membre de droite de cette inégalité étant un entier positif non nul, en prenant la partie entière, on a encore :

$$\left\lfloor\frac{n}{2^k}\right\rfloor < 2^{p-k+1}$$

1. La même démonstration permet aussi de la minorer et de trouver un encadrement de la complexité. Le lecteur est invité à faire ce calcul.

de sorte que :

$$C(n) < \gamma' p + \beta' \sum_{k=0}^{p-1} 2^{p-k+1} = \gamma' p + 2\beta' \sum_{k=0}^{p-1} 2^{p-k} = \gamma' p + 2\beta' \sum_{j=1}^{p} 2^j = \gamma' p + 4\beta' \sum_{j=0}^{p-1} 2^j = \gamma' p + 4\beta'(2^p - 1)$$

Or $2^p \leqslant n$. Ainsi :

$$C(n) < \gamma' p + 4\beta'(n-1)$$

D'où une complexité en $O(n)$.

Exercice 3 (Multiplication russe).

1. Cours : réécrire l'algorithme de multiplication russe sous forme récursive **terminale** en OCaml.
2. Prouver la terminaison de cet algorithme.
3. Démontrer que :
$$\forall x \in \mathbb{R},\ \forall q \in \mathbb{N}^*,\ \left\lfloor \frac{\lfloor x \rfloor}{q} \right\rfloor = \left\lfloor \frac{x}{q} \right\rfloor$$
4. En déduire une formule explicite pour le variant choisi.
5. Combien d'appels récursifs sont nécessaires pour que l'algorithme termine ?
6. Qu'en déduire sur la complexité temporelle de cet algorithme ?

Corrigé de l'exercice 3. *[Retour à l'énoncé]*

1. Voici une implémentation de l'algorithme de multiplication russe sous forme récursive terminale en OCaml

```
# let mult_russe p0 q0 =
  let rec aux acc p q =
    match p with
    | 0 -> acc
    | _ -> aux (acc + q*(p mod 2)) (p/2) (q*2)
  in aux 0 p0 q0;;
val mult russe : int -> int -> int = <fun>
```

Cette implémentation utilise un accumulateur `acc` pour transmettre à travers la chaîne d'appels le travail de multiplication effectué par les appels récursifs, sans devoir garder en mémoire ces informations dans des blocs d'activation empilés. Cela permet de réutiliser le même bloc d'activation pour tous les appels et d'éviter ainsi un empilement excessif de blocs d'activation pouvant mener à un effondrement de pile (*stack overflow*). L'algorithme est en fait codé dans une fonction auxiliaire `aux` gérant cet accumulateur, cette fonction auxiliaire étant ensuite appelée en initialisant l'accumulateur à 0 pour le premier appel.

2. Pour prouver la terminaison, on utilise, comme toujours, la technique du variant. Comme l'algorithme est implémenté de manière récursive, la suite qui fera office de variant a de forte chance d'être naturellement définie par récurrence.
On introduit donc la suite $(p_k)_{\mathbb{N}}$:
$$\begin{cases} p_0 \\ p_{k+1} = \lfloor \frac{p_k}{2} \rfloor \end{cases}$$
k représente le numéro d'appel imbriqué.
Il s'agit exactement du même variant que celui utilisé pour la preuve de terminaison de l'algorithme d'exponentiation rapide. La preuve de terminaison est donc identique, et on prouve ainsi que l'algorithme de multiplication russe se termine : il existe un appel k_0 pour lequel le cas de base est atteint : $p_{k_0} = 0$. Bien sûr, k_0 dépend de la valeur d'entrée p_0 !
3. Nous allons donc démontrer la propriété suivante :

Propriété 1

$$\forall x \in \mathbb{R},\ \forall q \in \mathbb{N}^*,\ \left\lfloor \frac{\lfloor x \rfloor}{q} \right\rfloor = \left\lfloor \frac{x}{q} \right\rfloor$$

Démonstration

Soit $q \geq 1$. Écrivons la division euclidienne de $\lfloor x \rfloor \in \mathbb{N}$ par q :

$$\exists a \in \mathbb{N} \text{ et } r \in \mathbb{N} \text{ tels que } \lfloor x \rfloor = a\,q + r \text{ et } 0 \leq r < q$$

On a donc :

$$\frac{\lfloor x \rfloor}{q} = \underbrace{a}_{\in \mathbb{N}} + \underbrace{\frac{r}{q}}_{0 \leq \frac{r}{q} < 1}$$

donc, par définition de la partie entière :

$$\left\lfloor \frac{\lfloor x \rfloor}{q} \right\rfloor = a$$

Démontrer la propriété revient donc à démontrer que :

$$a = \left\lfloor \frac{x}{q} \right\rfloor$$

Par définition de la partie entière de x :

$$\begin{aligned}
& \lfloor x \rfloor \leq x < \lfloor x \rfloor + 1 \\
\Leftrightarrow \quad & a\,q + r \leq x < a\,q + r + 1 && \text{en remplaçant } \lfloor x \rfloor \text{ par } aq + r \\
\Leftrightarrow \quad & a + \underbrace{\frac{r}{q}}_{\geq 0} \leq \frac{x}{q} < a + \underbrace{\frac{r+1}{q}}_{\leq 1} && \text{en divisant l'encadrement par } q > 0 \\
\Leftrightarrow \quad & a \leq a + \frac{r}{q} \leq \frac{x}{q} < a + \frac{r+1}{q} \leq a + 1 \\
\Rightarrow \quad & a \leq \frac{x}{q} < a + 1
\end{aligned}$$

Comme $a \in \mathbb{N}$, cela signifie exactement que :

$$\left\lfloor \frac{x}{q} \right\rfloor = a$$

et la propriété est démontrée.

4. Nous allons démontrer la formule explicite suivante pour la suite $(p_k)_{\mathbb{N}}$ par récurrence :

$$\forall k \in \mathbb{N},\ p_k = \left\lfloor \frac{p_0}{2^k} \right\rfloor$$

Cas de base : Pour $k = 0$, la formule est vraie car

$$\left\lfloor \frac{p_0}{2^0} \right\rfloor = \left\lfloor \frac{p_0}{1} \right\rfloor = \lfloor p_0 \rfloor = p_0$$

et, pour la dernière égalité, on a utilisé le fait que la valeur d'entrée de l'algorithme p_0 est un entier

Hérédité : Supposons la formule vraie au rang $k \geq 1$. La formule de récurrence définissant la suite nous donne :

$$p_{k+1} = \left\lfloor \frac{p_k}{2} \right\rfloor = \left\lfloor \frac{\left\lfloor \frac{p_0}{2^k} \right\rfloor}{2} \right\rfloor$$

En appliquant la propriété démontrée en question 3 pour $x = \dfrac{p_0}{2^k} \in \mathbb{R}$ et $q = 2 \in \mathbb{N}^*$, on obtient :

$$p_{k+1} = \left\lfloor \frac{p_k}{2} \right\rfloor = \left\lfloor \frac{\left\lfloor \frac{p_0}{2^k} \right\rfloor}{2} \right\rfloor = \left\lfloor \frac{\frac{p_0}{2^k}}{2} \right\rfloor = \left\lfloor \frac{p_0}{2^{k+1}} \right\rfloor$$

et on donc bien montré la propriété au rang $k + 1$. ■

5. Détermine le nombre exact de tours effectués revient à déterminer k_0, indice pour lequel la suite $(p_k)_{\mathbb{N}}$ devient nulle (stationnaire). Nous allons montrer rigoureusement que :

Propriété 2

Le nombre de tours de boucles effectués dans l'algorithme de la multiplication russe dépend de la première entrée p_0 et vaut :

$$k_0 = 1 + \lfloor \log_2(p_0) \rfloor$$

On peut sans restriction supposer que p_0 est positif. En effet, s'il ne l'est pas, soit q_0 est positif et on reporte le signe moins sur q_0. Soit q_0 est négatif, et, comme les deux signes moins se compensent dans la multiplication, on peut les ignorer.

Comme n'importe quel nombre entier positif, la valeur initiale p_0 peut être encadrée par deux puissances de 2 consécutives :

$$\exists m \in \mathbb{N},\ 2^m \leq p_0 < 2^{m+1}$$

En divisant l'encadrement par 2^m (ce qui est légal et ne change pas le sens des inégalités car $2^m > 0$), on en déduit que : $1 \leq \dfrac{p_0}{2^m} < 2$

Par définition de la partie entière, on en déduit que : $\left\lfloor \dfrac{p_0}{2^m} \right\rfloor = 1 \Leftrightarrow p_m = 1$

De même, en divisant cette fois l'encadrement par 2^{m+1}, on en déduit que : $\dfrac{1}{2} \leq \dfrac{p_0}{2^{m+1}} < 1$.

Par définition de la partie entière, on en déduit que : $\left\lfloor \dfrac{p_0}{2^{m+1}} \right\rfloor = 0 \Leftrightarrow p_{m+1} = 0$

Ainsi, $p_m = 1$ et $p_{m+1} = 0$.

$m+1$ est donc le premier indice pour lequel la suite $(p_k)_{\mathbb{N}}$ est stationnaire et atteint sa valeur limite 0 : c'est donc k_0 !

$$k_0 = m + 1$$

Revenons maintenant la manière dont nous avons défini m.

$$\begin{array}{lllll} & 2^m & \leq p_0 & < 2^{m+1} & \\ \Leftrightarrow & m\ln(2) & \leq \ln(p_0) & < (m+1)\ln(2) & \text{en prenant le logarithme, qui est une fonction strictement croissante} \\ \Leftrightarrow & m & \leq \log_2(p_0) & < m+1 & \text{en divisant l'encadrement par } \ln(2) > 0 \end{array}$$

et par définition de la partie entière, en sachant de plus que m est un entier par définition, on en déduit que :

$$m = \lfloor \log_2(p_0) \rfloor$$

En rassemblant l'ensemble des éléments démontrés, on en déduit que :

$$k_0 = 1 + m = 1 + \lfloor \log_2(p_0) \rfloor$$

Le nombre de tours de boucles effectués est donc égal à $1 + \lfloor \log_2(p_0) \rfloor$

6. Maintenant que nous savons que le nombre de tours de boucles est $1 + \lfloor \log_2(x_0) \rfloor$, on peut compter le nombre d'opérations élémentaires dans le pire des cas, qui correspond au cas où le test dans la boucle est toujours vrai :

$$3 \text{ affectations} + (1 + \lfloor \log_2(x_0) \rfloor) \times (2 \text{ tests} + 4 \text{ opérations})$$

Plus p_0 est un grand nombre, plus l'algorithme sera coûteux en nombre d'opérations. . On a même un modèle de croissance de cette complexité : la complexité temporelle évolue asymptotiquement de la même manière que $\log_2(p_0)$. On dit que l'algorithme a une **complexité temporelle logarithmique**.

Exercice 4 (Complexités récursives).
On considère l'algorithme d'exponentiation rapide **récursif**.

1. Écrire rapidement l'algorithme en OCaml
2. On appelle $C(n)$ la complexité (nombre de multiplications) pour un exposant n. Donner une relation de récurrence sur $C(n)$.
3. On se restreint au cas où n est une puissance de 2. Donner alors une expression de $C(n)$
4. On traite maintenant le cas général. Pour cela, on va traiter simultanément toutes les valeurs d'exposant n situées entre les mêmes puissances de 2 : $2^k \leq n < 2^{k+1}$. Conjecturer un encadrement de $C(n)$ pour ces valeurs de n.
5. Prouver cet encadrement pour toutes les valeurs d'exposant possibles
6. Que peut-on en conclure sur la complexité de l'algorithme d'exponentiation rapide ?

Corrigé de l'exercice 4. *[Retour à l'énoncé]*

1. L'algorithme d'exponentiation rapide repose sur la formule :

$$x^n = \begin{cases} x^{2k} & = (x^k)^2 & \text{si } n = 2k \\ x^{2k+1} & = (x^k)^2 \times x & \text{si } n = 2k+1 \end{cases}$$

La version récursive est immédiate à implémenter car très proche de la formulation mathématique :

```
# let rec exp_rapide_rec x n =
  match (n, n mod 2) with
  | (0, _) -> 1.0
  | (n, _) when n < 0  -> failwith "Valeur de n invalide (negative)"
  | (_, 0)  -> ( exp_rapide_rec x (n/2) ) ** 2.0
  |  _   -> ( exp_rapide_rec x (n/2) ) ** 2.0 *. x;;
val exp_rapide_rec : float -> int -> float = <fun>
# exp_rapide_rec 2.0 (-1);;
Exception: Failure "Valeur de n invalide (negative)".
# exp_rapide_rec 2.0 3;;
- : float = 8.
# exp_rapide_rec (-1.5) 4;;
- : float = 5.0625
```

2. On évalue la complexité temporelle de l'algorithme en comptant le nombre de multiplications. On rappelle que mettre au carré revient à multiplier le nombre par lui-même, donc à effectuer une seule multiplication. Dans ce cas, la complexité temporelle de l'algorithme s'exprime en fonction de l'entier n donné en entrée de l'algorithme comme :

$$C(0) = 0$$

$$C(n) = \begin{cases} 1 + C(\lfloor \frac{n}{2} \rfloor) & \text{si } n \text{ est pair} \\ 2 + C(\lfloor \frac{n}{2} \rfloor) & \text{si } n \text{ est impair} \end{cases}$$

3. On se restreint au cas où n est une puissance de 2. On écrit donc $n = 2^k$. La relation de récurrence devient alors :

$$\begin{aligned} C(2^0) &= C(1) = 2 \\ C(2^k) &= 1 + C(2^{k-1}) \text{pour } k \geq 1 \end{aligned}$$

Si on note $(u_k)_{k\in\mathbb{N}}$ la suite de terme général $u_k = C(2^k)$, alors :

$$\begin{aligned} u_0 &= 2 \\ u_k &= 1 + u_{k-1} \text{pour } k \geq 1 \end{aligned}$$

Ainsi, $(u_k)_{k\in\mathbb{N}}$ est une suite arithmétique de raison 1. On peut donc donner sa formule explicite :

$$u_k = 2 + k$$

On en déduit que :

$$C(2^k) = 2 + k$$

Autrement dit :

$$C(n) = 2 + \log_2(n)$$

lorsque n est une puissance de 2.

4. Soit n un entier tel que :

$$2^k \leq n < 2^{k+1}$$

Pour conjecturer un encadrement de $C(n)$, on peut commencer par s'appuyer sur le cas particulier de la question précédente. A chaque appel récursif, on fait soit une seule, soit 2 multiplication.

— Si n est une puissance de 2, alors, à chaque appel récursif, l'entier donné en entrée sera toujours pair : on ne fera donc qu'une seule multiplication à chaque appel, c'est le meilleur des cas possibles. Si n est tel que $2^k \leq n < 2^{k+1}$, la seule possibilité pour que n soit une puissance de 2 est que $n = 2^k$ et nous avons vu dans ce cas que la complexité est $k + 2$. Pour tous les autres nombres n vérifiant l'encadrement, la complexité sera plus grande. On a donc déjà :

$$k + 2 \leq C(n)$$

— A l'inverse, si la décomposition en produits de facteurs premiers de n ne fait pas apparaître de puissance de 2, alors à chaque appel récursif, le nombre entier donné en entrée sera impair : on fera donc systématiquement 2 multiplications à chaque appel récursif, c'est le pire des cas possibles. Donnons un petit exemple. Prenons le nombre $n = 15$. On a $2^3 \leq 15 < 2^4$. Voici la suite des valeurs d'entrée pour le paramètre n pour chaque appel récursif :

$$15 \to 7 \to 3 \to 1 \to 0$$

On obtient que des puissances impaires, ce qui nous amènera à effectuer un total de $2 \times 4 = 8$ multiplications en « remontant » les appels :

$$((x^2 \times x)^2 \times x)^2 \times x \leftarrow (x^2 \times x)^2 \times x \leftarrow x^2 \times x \leftarrow 1^2 \times x \leftarrow 1$$

Si $2^k \leq n < 2^{k+1}$, on a donc au maximum $(k+1)$ appels récursif et donc $2 \times (k+1)$ multiplications dans le pire des cas, si on a à chaque fois une entrée impaire.

$$C(n) \leq 2\,(k + 1)$$

On conjecture donc l'encadrement : Pour tout n un entier tel que $2^k \leq n < 2^{k+1}$, la complexité temporelle est telle que :

$$k + 2 \leq C(n) \leq 2\,(k + 1)$$

5. Pour prouver de manière rigoureuse cet encadrement, nous allons réaliser une preuve par récurrence sur l'entier k. En d'autres termes, nous allons prouver cet encadrement par « tranches » de puissances de 2.

$\mathcal{P}(k)$: Pour tout n un entier tel que $2^k \leq n < 2^{k+1}$, la complexité temporelle est telle que :

$$k + 2 \leq C(n) \leq 2\,(k + 1)$$

Initialisation : $k = 0$, et on considère tous les entiers n tels que $1 \leq n < 2$: il n'y en a qu'un seul, $n = 1$! On sait que $C(1) = 2$ ce qui est compatible avec l'encadrement :

$$k + 2 \leq C(n) \leq 2\,(k + 1)$$

qui donne, pour $k = 0$:

$$2 \leq C(n) \leq 2$$

Hérédité : Supposons la propriété vraie au rang k. Nous allons démontrer la propriété au rang $k+1$. Soit n un entier compris dans la « tranche » $k+1$, c'est-à-dire tel que : $2^{k+1} \leq n < 2^{k+2}$. Dans ce cas, en divisant par 2 l'encadrement, on en déduit que l'entier $\lfloor \frac{n}{2} \rfloor$ est compris dans l'intervalle :

$$2^k \leq \lfloor \frac{n}{2} \rfloor < 2^{k+1}$$

On peut donc lui appliquer l'hypothèse de récurrence :

$$k + 2 \leq C(\lfloor \frac{n}{2} \rfloor) \leq 2\,(k + 1)$$

Nous avons donné à la question 1 la formule de récurrence définissant $C(n)$:

$$C(0) = 0$$

$$C(n) = \begin{cases} 1 + C(\lfloor \frac{n}{2} \rfloor) & \text{si } n \text{ est pair} \\ 2 + C(\lfloor \frac{n}{2} \rfloor) & \text{si } n \text{ est impair} \end{cases}$$

On en déduit donc que :

$$1 + C(\lfloor \frac{n}{2} \rfloor) \leq C(n) \leq 2 + C(\lfloor \frac{n}{2} \rfloor)$$

En combinant avec l'inégalité **5.** :

$$1 + (k+2) \leq C(n) \leq 2 + 2\,(k+1)$$

$$\Leftrightarrow (k+1) + 2 \leq C(n) \leq 2\,((k+1)+1)$$

On a donc bien montrer que la propriété était vraie au rang $k+1$.

Conclusion : Par principe de récurrence, on en déduit donc que la propriété est vraie pour toutes les tranches $[\![2^k,\, 2^{k+1}-1]\!]$, et donc pour, pour n'importe quel n compris dans n'importe quelle tranche $2^k \leq n < 2^{k+1}$, on a :

$$k + 2 \leq C(n) \leq 2\,(k+1)$$

Soit n tel que $2^k \leq n < 2^{k+1}$ où $k \in \mathbb{N}$. Alors :

$$\begin{aligned} & 2^k \leq n < 2^{k+1} \\ \Leftrightarrow \quad & k\ln(2) \leq \ln(n) < (k+1)\ln(2) \\ \Leftrightarrow \quad & k \leq \log_2(n) < k+1 \\ \Leftrightarrow \quad & k = \lfloor \log_2(n) \rfloor \end{aligned}$$

En transformant l'inégalité prouvée à la question précédente, on a donc :

$$\log_2(n) + 2 \leq C(n) \leq 2\,(\log_2(n) + 1)$$

On en conclut donc que $C(n) \in O(\log_2(n))$. L'algorithme d'exponentiation rapide récursif est de **complexité temporelle logarithmique**.

En fait, on a même mieux car la complexité n'est pas seulement dominée par un logarithme, elle est **encadrée** par des logarithmes. On dit que $C(n) \in \Theta(\log_2(n))$:$C(n)$ est un θ de $\log_2(n)$. [2]

2. Attention toutefois, ce n'est pas un équivalent à cause des facteurs différents devant les deux logarithmes base 2 encadrant $C(n)$.

Exercice 5 (Recherche dichotomique).
On considère ci-dessous une implémentation possible de l'algorithme de recherche dichotomique.

```
let recherche_dichotomique v t =

  let l = ref 0 and
      r = ref ( Array.length t ) and
      trouve   = ref false and
      idx_loc = ref (-1) in

  while (l < r && !trouve = false) do

    let m = ((!r) + (!l) )/2 in
    let val_milieu = t.(m) in

    if (val_milieu = v) then
      (
        trouve := true;
        idx_loc := m
      )
    else
      (
        if (val_milieu > v) then
          r := m-1
        else
          l := m+1
      )
  done;
  !idx_loc
;;
```

1. Donner la complexité spatiale de cette fonction
2. Montrer que la complexité temporelle dans le pire des cas est un $O(n)$
3. Cette borne n'est cependant pas très serrée. On peut dire beaucoup mieux. Établir une relation de récurrence sur la taille de la fenêtre de recherche.
4. En déduire une majoration de la taille de cette fenêtre de recherche au cours de l'algorithme
5. En déduire que la complexité dans le pire cas est en fait en $O(\log_2 n)$
6. On introduit la notation Θ. On dit qu'une fonction $g : \mathbb{N} \to \mathbb{R}^+$ est un Θ de $f : \mathbb{N} \to \mathbb{R}^+$, et l'on note $g(n) \in \Theta(f(n))$ s'il existe deux facteurs k_1, $k_2 \in \mathbb{R}^+$ et un rang $n_0 \in \mathbb{N}$ tels que

$$\forall n \geq n_0,\ k_1\, f(n) \leq g(n) \leq k_2\, f(n)$$

   Identifier un pire cas permettant de prouver que la complexité est en $\Theta(\log_2(n))$

Corrigé de l'exercice 5. *[Retour à l'énoncé]*

1. Pour évaluer la complexité spatiale, on ne prend en compte que la mémoire allouée pour le travail de l'algorithme. L'espace mémoire alloué pour le stockage des données d'entrée n'est pas compté. Ici, seules quelques variables de travail (`l`, `r`,`trouve`, `idx_loc`, `val_milieu`, `m`) sont nécessaires au travail de l'algorithme. La complexité spatiale est donc très faible, et indépendante de la taille du tableau d'entrée. C'est un $O(1)$.
2. Dans le pire cas imaginable, on effectue un test de comparaison avec la valeur recherchée sur toutes les cases du tableau. Cela correspond au pire cas de l'algorithme naïf de recherche séquentielle. Dans tous les cas, nous sommes donc certains que l'algorithme de recherche dichotomique a une complexité temporelle $C(n) \in O(n)$.

3. En fait, dans l'algorithme de recherche dichotomique, ce pire cas n'arrive jamais. Nous allons le montrer dans la suite de l'exercice. On commence par établir une relation de récurrence donnant la taille de la fenêtre de recherche. On introduit pour commencer 3 suites :

- $(l_k)_{k\in\mathbb{N}}$ la suite des valeurs prises par la variable `l` (*left*) représentant la borne gauche de la fenêtre de recherche.
- $(r_k)_{k\in\mathbb{N}}$ la suite des valeurs prises par la variable `r` (*right*) représentant la borne droite de la fenêtre de recherche.
- $(m_k)_{k\in\mathbb{N}}$ la suite des valeurs prises par la variable `m` (*middle*) représentant le centre de la fenêtre de recherche.

En analysant le code proposé, on obtient les relations de récurrence suivantes :

$$\begin{aligned} l_0 &= 0 \\ l_{k+1} &= \begin{cases} m_k + 1 & \text{si } v > m_k \\ l_k & \text{sinon} \end{cases} \end{aligned}$$

$$\begin{aligned} r_0 &= n \\ r_{k+1} &= \begin{cases} r_k & \text{si } v > m_k \\ m_k - 1 & \text{sinon} \end{cases} \end{aligned}$$

Et on a toujours :

$$m_k = \lfloor \frac{l_k + r_k}{2} \rfloor$$

La taille de la fenêtre de recherche peut elle aussi être représentée par une suite $(d_k)_{k\in\mathbb{N}}$:

$$d_k = r_k - l_k$$

On a :

$$\begin{aligned} d_0 &= n - 0 = n \\ d_{k+1} &= \begin{cases} r_k - m_k - 1 = r_k - m_k - 1 & \text{si } v > m_k \\ m_k - 1 - l_k = m_k - l_k - 1 & \text{sinon} \end{cases} \end{aligned}$$

4. Nous allons montrer que $d_k \leq \frac{n}{2^k}$. Par définition de la partie entière, et en lien avec la définition de la suite (m_k), on a :

$$\frac{r_k + l_k}{2} - 1 < m_k \leq \frac{r_k + l_k}{2}$$

Si $v > m_k$ **:** $d_{k+1} = r_k - m_k - 1$. On renverse l'encadrement **4.** (multiplication par -1) :

$$-\frac{r_k + l_k}{2} \leq -m_k < -\frac{r_k + l_k}{2} + 1$$

Et on ajoute r_k à tous les membres :

$$\begin{aligned} & r_k - \frac{r_k + l_k}{2} \leq r_k - m_k < r_k - \frac{r_k + l_k}{2} + 1 \\ \Leftrightarrow \quad & \frac{r_k - l_k}{2} \leq r_k - m_k < \frac{r_k - l_k}{2} + 1 \\ \Leftrightarrow \quad & \frac{d_k}{2} - 1 \leq d_{k+1} < \frac{d_k}{2} \end{aligned}$$

Si $v < m_k$ **:** $d_{k+1} = m_k - l_k - 1$, et en utilisant l'encadrement **4.** :

$$\begin{aligned} & \frac{r_k + l_k}{2} - 1 - l_k < m_k - l_k \leq \frac{r_k + l_k}{2} - l_k \\ \Leftrightarrow \quad & \frac{r_k - l_k}{2} - 1 < m_k - l_k \leq \frac{r_k - l_k}{2} \\ \Leftrightarrow \quad & \frac{d_k}{2} - 2 < d_{k+1} \leq \frac{d_k}{2} - 1 < \frac{d_k}{2} \end{aligned}$$

Dans tous les cas, on a donc

$$d_{k+1} < \frac{d_k}{2} \tag{1}$$

Et par une récurrence immédiate :

$$d_k < \frac{d_0}{2^k} \Leftrightarrow d_k < \frac{n}{2^k}$$

5. L'algorithme s'arrête lorsque $d_k = 0$. Comme (d_k) est une suite à valeurs entières, cela advient dès que $\frac{n}{2^k} < 1$.

$$\begin{aligned} & \frac{n}{2^k} < 1 \\ \Leftrightarrow \quad & n < 2^k \\ \Leftrightarrow \quad & \log_2(n) < k \\ \Leftrightarrow \quad & k = \lfloor \log_2(n) \rfloor + 1 \end{aligned}$$

On a donc au maximum $k = \lfloor \log_2(n) \rfloor + 1$ itérations et donc la complexité temporelle est en $O(\log_2(n))$

6. Pour montrer que la complexité au pire est un *theta* de $\log_2(n)$, il suffit de montrer qu'il existe effectivement au moins une situation dans laquelle cette complexité au pire est atteinte. C'est bien entendu le cas lorsque la valeur v recherchée n'est pas présente dans le tableau.