Séquence 7 - Analyse d'algorithmes - Preuves

Tout algorithme nécessite une étude théorique afin d'étudier :

sa terminaison : est-ce que l'algorithme se termine et rendre un résultat en temps fini ?

sa correction : est-ce que l'algorithme retourne un résultat conforme à la spécification ayant guidé sa conception ?

sa complexité : quel est le coût de cet algorithme en terme de temps de calcul (complexité temporelle) et en terme d'utilisation des ressources mémoires (complexité spatiale) ? Est-il meilleur de ce point de vue que d'autres algorithmes répondant aux mêmes spécifications ?

Dans cette séquence, nous allons découvrir différents outils théoriques qui permettent de répondre de manière rigoureuse, par des démonstrations mathématiques, à ces questions.

I. Préliminaires mathématiques

I. 1. Partie entière

Définition 1 (Partie entière)

La partie entière d'un nombre $x > 0$ réel est l'unique nombre $n \in \mathbb{Z}$ vérifiant :

$$n \leq x < n + 1$$

C'est l'entier inférieur à x le plus proche de x.

Notation 1

On notera $n = \lfloor x \rfloor$ la partie entière de x.

Propriété 1

On a donc également, $\forall x \in \mathbb{R}$:

$$x - 1 < \lfloor x \rfloor \leq x$$

I. 2. Propriétés de $\mathbb{N}$

Théorème 1 (Plus petit élément)

Tout ensemble E **non vide** inclus dans $\mathbb{N}$ admet un plus petit élément :

$$\forall E \subset \mathbb{N}, \exists n_0 \in E\,, \forall n \in E,\, n_0 \leq n$$

Théorème 2 (Plus grand élément)

Tout ensemble E **non vide borné** inclus dans $\mathbb{N}$ est fini et admet un plus grand élément :

$$\forall E \subset \mathbb{N},\, \exists M > 0,\, \forall n \in E,\, n \leq M \,\exists\, n_{\max} \in E\,, \forall n \in E,\, n \leq n_{\max}$$

I. 3. Raisonnements par récurrence

Théorème 3 (Principe de récurrence simple)

Soit $\mathcal{P}(n)$ une propriété dépendant d'un paramètre entier $n \in \mathbb{N}$. Si on arrive à montrer que :

- $\mathcal{P}(0)$ est vraie (cas de base, aussi appelé initialisation)
- Pour $n \in \mathbb{N}$, $\mathcal{P}(n) \Rightarrow \mathcal{P}(n+1)$ (hérédité)

alors $\mathcal{P}(n)$ est valide pour tout $n \in \mathbb{N}$.

Démonstration

Raisonnons par l'absurde et supposons que sous les hypothèses du théorème, la conclusion du théorème est fausse.
On considère l'ensemble $E = \{n \in \mathbb{N},\, \mathcal{P}(n) \text{ est fausse}\} \subset \mathbb{N}$.
Si le théorème est faux, alors E est non vide. De plus, E est un sous-ensemble de $\mathbb{N}$. Il admet donc un plus petit élément d'après 1. On note $n_0 \in E$ ce plus petit élément.
n_0 est différent de 0 car sinon, cela contredirait le cas de base du théorème. Donc $n_0 - 1 \in \mathbb{N}$ et $\mathcal{P}(n_0 - 1)$ est vraie par minimalité de n_0.
Mais alors, d'après l'hypothèse d'hérédité , $\mathcal{P}(n_0)$ doit aussi être vraie, ce qui est absurde car $n_0 \in E$ et donc par définition de E, $\mathcal{P}(n_0)$ est fausse : on aboutit donc à une contradiction.
Donc la conclusion du théorème est vraie.

Théorème 4 (Principe de récurrence forte)

Soit $\mathcal{P}(n)$ une propriété dépendant d'un paramètre entier $n \in \mathbb{N}$. Si on arrive à montrer que :

$$\forall k < n,\, \mathcal{P}(k) \text{ est vraie} \Rightarrow \mathcal{P}(n) \text{ est vraie}$$

alors $\mathcal{P}(n)$ est vraie pour tout $n \in \mathbb{N}$

Démonstration

Il suffit d'appliquer le principe de récurrence faible à la propriété :

$$\mathcal{P}'(n) : \quad \forall k < n,\ \mathcal{P}(k) \text{ est vraie}$$

I. 4. Suites à valeurs entières

Définition 2 (Décroissance, stationnarité d'une suite)

Une suite $(u_n)_{\mathbb{N}}$ est **décroissante** si :

$$\forall\, n \in \mathbb{N},\ u_{n+1} \leq u_n \Leftrightarrow u_{n+1} - u_n \leq 0$$

Une suite $(u_n)_{\mathbb{N}}$ est **strictement décroissante** si :

$$\forall\, n \in \mathbb{N},\ u_{n+1} < u_n \Leftrightarrow u_{n+1} - u_n < 0$$

Une suite $(u_n)_{\mathbb{N}}$ est **stationnaire** si :

$$\exists n_0 \in \mathbb{N},\ \exists \ell,\ \forall n \geq n_0,\ u_n = \ell$$

Propriété 2

Soit une suite $(u_n)_{\mathbb{N}}$ telle que :

- $(u_n)_{\mathbb{N}}$ est à valeurs entières
- $(u_n)_{\mathbb{N}}$ est décroissante

Alors

- soit $(u_n)_{\mathbb{N}}$ est stationnaire à partir d'un certain rang :

$$\exists n_0 \in \mathbb{N},\ \exists \ell,\ \forall n \geq n_0,\ u_n = \ell$$

- soit $(u_n)_{\mathbb{N}}$ devient négative à partir d'un certain rang

$$\exists n_0 \in \mathbb{N},\ \forall n \geq n_0,\ u_n \leq 0$$

De plus, si $(u_n)_{\mathbb{N}}$ est strictement décroissante, alors seul le deuxième cas peut se produire.

Enfin, si $(u_n)_{\mathbb{N}}$ est à valeurs dans $\mathbb{N}$ (c'est à dire à valeurs entières positives) et si elle est strictement décroissante, elle ne peut l'être que jusqu'à un certain rang et elle devient ensuite stationnaire en 0. [a]

a. Cela vient du fait que l'ordre usuel sur $\mathbb{N}$ est un ordre bien fondé.

II. Preuves de terminaison et technique du variant

Définition 3 (Terminaison d'un algorithme)

Un algorithme se termine s'il **renvoie un résultat en temps fini** pour toutes les entrées possibles de l'algorithme.

Il existe deux situations dans lesquelles une preuve de terminaison est nécessaire :

en présence d'appels récursifs : il faut montrer que l'imbrication des appels récursifs aura une fin ;

en présence de boucles, en particulier de boucles `while` : il faut montrer que le test de boucle finira par devenir faux en temps fini.

II. 1. Technique du variant : principe de la méthode

La principale technique à votre disposition pour démontrer la terminaison d'un algorithme est la **technique du variant**.

Elle consiste à exhiber une suite, appelée variant, construite à partir des variables de l'algorithme et telle que :

- cette suite est à valeurs dans $\mathbb{N}$
- cette suite décroît strictement au cours de l'exécution de l'algorithme.

Cette suite doit en quelque sorte quantifier la distance, en terme de nombres d'appels récursifs, qui nous sépare encore du cas de base. Si l'algorithme a été bien conçu, on se rapproche de plus en plus du cas de base au cours des appels récursifs jusqu'à l'atteindre. Cela explique donc que l'on cherche une suite décroissante car cette distance au cas de base doit diminuer au cours de l'algorithme pour qu'il se termine.

La technique du variant repose sur le résultat fondamental suivant :

Propriété 3 ($(\mathbb{N}, \leq)$ est bien ordonné)

Il n'existe pas de suite infinie strictement décroissante (pour l'ordre usuel $\leq$ sur les entiers) dans $\mathbb{N}$.
On dit que l'ensemble $\mathbb{N}$, muni de la relation d'ordre usuelle $\leq$ sur les entiers est un ensemble bien fondé.

Méthodologie 1 (Méthode de démonstration).
La terminaison d'un algorithme se montre plutôt bien en effectuant un raisonnement par l'absurde, même si d'autres démonstrations sont tout à fait acceptables - nous en donnerons d'ailleurs des exemples. On procède en général comme ceci :

- On fait l'hypothèse que l'algorithme ne termine pas
- On construit une quantité entière positive $u_k \in \mathbb{N}$ à partir des quantités intervenant dans l'algorithme à l'appel récursif k, souvent les paramètres d'entrée de l'algorithme. L'indice de cette suite correspond au numéro d'appel récursif.
- On dit que, comme l'algorithme ne termine pas, il y a une infinité d'appels récursifs donc la suite des paramètres d'entrée possède une infinité de termes. On en déduit donc que la suite des valeurs u_k, qui a été construite à partir de ces entrées, est elle aussi infinie. On a donc une suite $(u_k)_{k\in\mathbb{N}}$ infinie, ayant une infinité de termes.
- On montre que cette suite est strictement décroissante, en raisonnant à partir de l'algorithme et de l'évolution des paramètres d'entrée dans les appels récursifs (on doit avoir un nombre fini d'appels récursif à chaque appel)
- On conclut par l'absurde car on aura alors exhibé une suite infinie strictement décroissante à valeurs dans $\mathbb{N}$, ce qui est impossible d'après la proposition précédente.

II. 2. Un premier exemple : algorithmes d'exponentiation

Vous avez vu et codé deux algorithmes, appelés algorithmes d'exponentiation, donc le but est de calculer x^n où x est une valeur réelle donnée et n un entier positif donné.

II. 2. a. Algorithme d'exponentiation naïf

Il repose sur la définition même de la notion de puissance entière positive d'un nombre :

$$x^n = x \times x^{n-1}$$

On peut en donner une version itérative et une version récursive, ici codées en OCaml :

```
# let exp_naive x n =

  assert (n >= 0);

  let r = ref 1.0 in
  let i = ref 0 in
  while (!i < n )
  do
    r := !r *. x;
    i := !i + 1
  done;
  !r;;
val exp_naive : float -> int -> float = <fun>
# exp_naive 2.0 (-1);;
Exception: Assert_failure ("//toplevel//", 60
# exp_naive 2.0 3;;
- : float = 8.
# exp_naive (-1.5) 4;;
- : float = 5.0625
```

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

II. 2. b. Terminaison de l'algorithme d'exponentiation naïf

Montrons que l'algorithme termine.
On raisonne **par l'absurde** et on suppose que l'algorithme ne termine pas. Cela signifie que les cas d'arrêt (cas de base $n = 0$ et cas d'erreur $n < 0$) ne sont jamais atteints. On note n_k la valeur du second paramètre d'entrée (exposant) à l'appel récursif numéro k. Comme on a supposé que l'algorithme ne termine pas, on en déduit à la fois que la suite des n_k est infinie (infinité de termes) et qu'elle est strictement positive (sinon, on aurait atteint un cas d'arrêt).
La suite des valeurs $(n_k)_{k\in\mathbb{N}}$ va nous servir de variant. On a déjà expliqué qu'elle était infinie et à valeurs dans $\mathbb{N}^*$. Il nous reste à montrer qu'elle est strictement décroissante pour obtenir une contradiction et conclure le raisonnement.
D'après l'algorithme, cette suite vérifie la relation de récurrence suivante :

$$\begin{cases} n_0 & = n \\ n_{k+1} & = n_k - 1 \end{cases}$$

La stricte décroissance est évidente (suite arithmétique de raison -1) :

$$n_{k+1} - n_k = -1 < 0$$

$(n_k)_{k\in\mathbb{N}}$ est donc une suite infinie strictement décroissante à valeurs dans $\mathbb{N}$: cela est impossible. Donc l'hypothèse de départ est fausse : l'algorithme se termine.

Pour la version itérative, le variant et la la logique de démonstration sont identiques.

II. 2. c. Algorithme d'exponentiation rapide

C'est un algorithme qui permet également de calculer une puissance entière d'un nombre réel mais de manière beaucoup plus efficace (nous allons bientôt voir pourquoi). Il 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
```

La version itérative requiert un peu plus d'explications. La division euclidienne de n par 2 donne :

$$n = (n \bmod 2) + k \times 2$$
$$x^n = x^{n \bmod 2} \times (x^2)^k$$

On peut à nouveau écrire la division euclidienne de k par 2 :

$$k = (k \bmod 2) + \lfloor \frac{k}{2} \rfloor \times 2$$

$$x^n = x^{n \bmod 2} \times (x^2)^{k \bmod 2} \times (x^4)^{\lfloor \frac{k}{2} \rfloor}$$

Et on peut à nouveau, en posant $k' = \lfloor \frac{k}{2} \rfloor$, faire la même opération :

$$x^n = x^{n \bmod 2} \times (x^2)^{k \bmod 2} \times (x^4)^{k' \bmod 2} \times (x^8)^{\lfloor \frac{k'}{2} \rfloor} \ldots \text{etc}$$

On va donc diviser par 2 l'exposant `k` à chaque itération et mettre à jour une variable `a` pour travailler d'abord sur x, puis sur x^2, puis sur x^4...etc

```
# let rec exp_rapide x n =

  let r = ref 1.0 in
  let k = ref n in
  let a = ref x in

  while (!k > 0)
  do
    if (!k mod 2 = 1) then
      r := !r *. !a;
    a := !a *. !a;

    k := !k / 2
  done;
  !r;;
val exp_rapide : float -> int -> float = <fun>
# exp_rapide 2.0 (-1);;
- : float = 1.
# exp_rapide 2.0 3;;
- : float = 8.
# exp_rapide (-1.5) 4;;
- : float = 5.0625
```

II. 2. d. Terminaison de l'algorithme d'exponentiation rapide

Montrons que l'algorithme termine.
On raisonne **par l'absurde** et on suppose que l'algorithme ne termine pas. Cela signifie que les cas d'arrêt (cas de base $n = 0$ et cas d'erreur $n < 0$) ne sont jamais atteints. On note n_k la valeur du second paramètre d'entrée (exposant) à l'appel récursif numéro k. Comme on a supposé que l'algorithme ne termine pas, on en déduit à la fois que la suite des n_k est infinie (infinité de termes) et qu'elle est strictement positive (sinon, on aurait atteint un cas d'arrêt).
La suite des valeurs $(n_k)_{k\in\mathbb{N}}$ va nous servir de variant. On a déjà expliqué qu'elle était infinie et à valeurs dans $\mathbb{N}^*$. Il nous reste à montrer qu'elle est strictement décroissante pour obtenir une contradiction et conclure le raisonnement.
D'après l'algorithme, cette suite vérifie la relation de récurrence suivante :

$$\begin{cases} n_0 & = n \\ n_{k+1} & = \left\lfloor \frac{n_k}{2} \right\rfloor \end{cases}$$

On a :

$$n_{k+1} - n_k = \left\lfloor \frac{n_k}{2} \right\rfloor - n_k$$

Or, $\left\lfloor \frac{n_k}{2} \right\rfloor \leq \frac{n_k}{2}$ par définition de la partie entière. Donc :

$$n_{k+1} - n_k \leq \frac{n_k}{2} - n_k = -\frac{n_k}{2}$$

Or, on a montré que $n_k \in \mathbb{N}^*$ donc $n_{k+1} - n_k < 0$ et la suite est strictement décroissante. $(n_k)_{k\in\mathbb{N}}$ est donc une suite infinie strictement décroissante à valeurs dans $\mathbb{N}$: cela est impossible. Donc l'hypothèse de départ est fausse : l'algorithme se termine.
La preuve pour la version itérative est tout à fait semblable, là encore avec le même variant !

II. 3. La terminaison des algorithmes : un problème parfois épineux !

Attention, il ne faut pas croire qu'on sache toujours prouver qu'une fonction termine ou non.

Conjecture de Goldbach. Par exemple, personne ne peut, à l'heure actuelle, prouver qu'une fonction qui s'arrêterait quand elle rencontre un nombre pair supérieur ou égal à 4 qui n'est pas somme de deux nombres premiers, s'arrête effectivement (la conjecture de Goldbach, non démontrée à ce jour, dit qu'une telle fonction ne termine pas).

Algorithme de Syracuse. Nul ne sait prouver que la fonction `syracuse` ci-dessous termine ! Sa terminaison n'est donc que conjecturée et n'a jamais été prouvée !

```c
unsigned int syracuse(unsigned int n)
{
  unsigned int u;
  unsigned int cnt;

  assert(n != 0);

  u = n;
  cnt = 0;

  while (u != 1)
  {
    if (u%2 == 0)
      u = u/2;
    else
      u = 3*u+1;

    cnt = cnt + 1;
  }

  return cnt;
}
```

A vous de tester : essayez de trouver une valeur d'entrée pour laquelle cet algorithme ne termine pas !

Indécidabilité du problème de terminaison. Il est même facile de prouver qu'il n'existera jamais d'algorithme permettant de savoir si une fonction récursive termine. En effet, supposons que ce soit le cas. On pourrait alors écrire une fonction`termine: 'a->'b->bool`, qui prendrait en argument une fonction f, et renverrait `true` si f termine, et `false` sinon.
Considérons alors la fonction :

```
let rec tordue ()= if (termine tordue) then tordue ();;
```

Alors si tordue termine, elle ne termine pas, et si elle ne termine pas, elle termine. C'est contradictoire, non ?

II. 4. Limites du cadre d'étude précédent

Dans certains cas, le cadre de démonstration précédent est trop limité pour nous permettre de démontrer proprement/facilement/élégamment la terminaison de l'algorithme.

Exemple 1 (Fonction d'Ackermann)

On considère la fonction d'Ackermann `ack` qui prend en paramètres deux entiers naturels n et m :

$$\begin{cases} \texttt{ack}(0,\, m) & = m+1 & \\ \texttt{ack}(n,\, 0) & = \texttt{ack}(n-1,\, 1) & \text{si } n > 0 \\ \texttt{ack}(n, m) & = \texttt{ack}(n-1,\, \texttt{ack}(n, m-1)) & \text{si } n > 0 \text{ et } m > 0 \end{cases}$$

Cette fonction se termine toujours (même si le temps d'exécution peut être très long !) et cela peut être démontré. Cette fonction est souvent utilisée pour illustrer la puissance des preuves de terminaison utilisant la notion d'ordre bien fondé, là où les preuves utilisant des variants à valeurs entières se révèlent longues et fastidieuses.
Ici, il y a plusieurs difficultés - la variation simultanée de plusieurs paramètres, des arguments d'appels récursifs qui sont eux-mêmes des appels récursifs - qui sont traitées avec élégance dans les preuves plus sophistiquées utilisant des ensembles bien ordonnés :

Nous présentons au second semestre une généralisation de la technique du variant permettant d'adresser ces difficultés.

III. Correction

Qu'un algorithme termine est une bonne chose... encore faut-il s'assurer qu'il renvoie une réponse juste, conforme aux attendus !

Définition 4 (Correction d'un algorithme)

Démontrer la correction d'un algorithme, c'est prouver qu'il renvoie une réponse juste, dans le cadre des spécifications ayant gouverné sa conception.

Nous commencerons donc par redonner quelques précisions sur la définition de ces attendus à travers la définition d'une **spécification** pour l'algorithme. Puis nous donnerons deux techniques de preuve de correction et quelques exemples d'applications. Attention cependant, il n'y a pas de recette miracle !

Définition 5 (Correction totale ou partielle)

- On dira que l'on a fourni une **preuve de correction totale** de l'algorithme si l'on a démontré sa correction mais aussi sa terminaison.
- Si l'on a réussi à prouver la correction d'un algorithme sans pouvoir prouver sa terminaison, on dira que l'on a fourni une **preuve de correction partielle**

III. 1. Spécification d'un algorithme

Un algorithme répond à un problème : à partir de certaines entrées, produire un certain résultat ou effet. Avant même de concevoir un algorithme, il faut énoncer clairement le problème posé. La spécification d'un algorithme décrit précisément à la fois l'ensemble des entrées acceptées par l'algorithme et ce que l'algorithme doit produire.

Définition 6 (Spécification d'un algorithme)

Donner la spécification d'un algorithme, c'est indiquer avec précision deux aspects :

En entrée : le nombre et la nature (type) des données d'entrée nécessaires à son fonctionnement ; si ces données d'entrée doivent vérifier certaines **préconditions** garantissant le bon fonctionnement de l'algorithme, il faut également le préciser ;

En sortie : le ou les résultat(s) attendu(s), en précisant leur(s) type(s) et en différenciant les cas si nécessaire.

Définition 7 (Préconditions)

Les **préconditions** sont des contraintes que doivent vérifier les valeurs fournies en entrée de l'algorithme pour que le bon fonctionnement soit garanti. Elle peuvent décrire des plages de valeurs d'entrée acceptables (positivité, intervalle, indice valide dans un tableau) ou des informations sur la structure des données fournies (par exemple pour une liste ou un tableau : ne pas être vide, être trié par ordre croissant, ne pas contenir de doublons...)

Remarque. Typiquement, en commentaire au dessus de la définition d'une fonction, il est intéressant de rappeler les préconditions s'appliquant sur les entrées... Cela pourra être utile à un programmeur reprenant et utilisant votre code, pour qu'il l'utilise en connaissance de cause. Par exemple, rajouter un commentaire en précisant que votre fonction de recherche dichotomique attend un tableau trié.

Attention. La spécification doit être réalisée et comprise avec soin. Elle constitue en effet une sorte de **contrat** : si un utilisateur ou un autre programmeur utilise notre algorithme en lui fournissant des données d'entrée vérifiant les contraintes énoncées dans les préconditions, il est en droit de s'attendre à obtenir le résultat décrit dans la spécification.

A l'inverse, si les entrées fournies par l'utilisateur ou le programmeur ne correspondent pas aux critères énoncés dans les spécifications, notre algorithme n'aura peut-être pas le comportement attendu et l'utilisateur en sera pour ses frais. Toutefois, dans ce dernier cas, mieux vaut rester *fair-play* et appliquer les principes de la **programmation défensive** en récupérant les erreurs récupérables (exceptions) [a] ou en arrêtant proprement le code en cas de préconditions non vérifiées (`assert` ou message d'erreur constructif).

La spécification sert aussi de boussole pour démontrer la correction d'un algorithme. En effet, pour prouver qu'un algorithme fournit les réponses attendues, encore faut-il avoir décrit avec précision en quoi consiste un comportement correct de la part de cet algorithme.

a. On parle de code **résilient**

Notation 2

Dans la suite, on notera x la valeur contenue dans une variable `x`.

III. 2. 1ère technique : raisonnement équationnel et preuve par récurrence

Le **raisonnement équationnel avec preuve par récurrence** est une technique de preuve de correction qui s'applique généralement pour les **algorithmes récursifs ne modifiant aucune donnée**. Elle est tout **particulièrement indiqué pour les codes récursifs implémentés avec des variables immuables**, comme c'est le cas par défaut en OCaml lorsque l'on n'utilise pas de références.

III. 2. a. Preuve de correction de l'exponentiation naïve récursive

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

Nous allons utiliser une **récurrence simple** pour démontrer la correction de l'algorithme (et nous aurons ainsi montré sa correction totale grâce à la preuve de terminaison fournie précédemment !)

Démonstration

On note $\mathcal{P}(n)$: « Pour tout réel `x`, `exp_naive_rec x n` renvoie la valeur x^n »

Cas de base : $\mathcal{P}(0)$ est vraie car l'algorithme est tel que l'évaluation de (`exp_naive_rec x 0`) retourne la valeur 1, qui est bien égale à x^0 pour n'importe quelle valeur de x

Hérédité : Supposons que $\mathcal{P}(n)$ est vraie. Fixons une valeur réelle x. Alors l'appel `exp_naive_rec x (n+1)` renvoie :

$$\begin{aligned} \texttt{exp\_naive\_rec x (n+1)} &= x \times (\texttt{exp\_naive\_rec x (n+1-1)}) && \text{d'après l'algorithme} \\ &= x \times (\texttt{exp\_naive\_rec x n}) && \text{d'après l'algorithme} \\ &= x \times x^n && \text{par hyp. rec.} \\ &= x^{n+1} \end{aligned}$$

donc $\mathcal{P}(n+1)$ est vraie.

Ainsi, $\mathcal{P}(n)$ est vraie pour toute entrée n entière positive choisie par l'utilisateur : notre code d'exponentiation naïve est correct.

III. 2. b. Preuve de correction de l'exponentiation rapide récursive

```
# 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
```

Nous allons utiliser une **récurrence** forte car l'algorithme fait appel à des valeurs qui ne sont pas des prédécesseurs immédiats de n [1]

1. C'est généralement la bonne option pour les algorithmes dichotomiques...

Démonstration

On note $\mathcal{P}(n)$: « Pour tout réel `x`, `exp_rapide_rec x n` renvoie la valeur x^n »

Cas de base : $\mathcal{P}(0)$ est vraie car l'algorithme est tel que l'évaluation de (`exp_rapide_rec x 0`) retourne la valeur 1, qui est bien égale à x^0 pour n'importe quelle valeur de x

Hérédité : Supposons que, pour tout entier positif $k < n$, $\mathcal{P}(k)$ est vraie. On fixe la valeur réelle x de `x`. On veut montrer que $\mathcal{P}(n)$ est vraie. On distingue deux cas :

Si n est pair : on peut alors écrire $n = 2k$ avec k un entier et $k < n$.

$$\begin{aligned}
\texttt{exp_rapide_rec x n} &= (\texttt{exp_rapide_rec x n/2})^2 && \text{(3ème cas du filtrage)}\\
&= (\texttt{exp_rapide_rec x k})^2 && \text{(div euclid) renvoie } k\\
&= \left(x^k\right)^2 && \text{par h.r. forte } \forall k < n\\
&= x^{2k}\\
&= x^n && \text{car } 2k = n
\end{aligned}$$

et $\mathcal{P}(n)$ est bien vérifiée.

Si n est impair : on peut alors écrire $n = 2k + 1$ avec k un entier et $k < n$.

.

$$\begin{aligned}
\texttt{exp_rapide_rec x n} &= (\texttt{exp_rapide_rec x n/2})^2 && \text{(4ème cas du filtrage)}\\
&= (\texttt{exp_rapide_rec x k})^2 \times x && \text{(div euclid) renvoie } k\\
&= \left(x^k\right)^2 && \times x \text{ par h.r. forte } \forall k < n\\
&= x^{2k} \times x\\
&= x^{2k+1}\\
&= x^n && \text{car } 2k + 1 = n
\end{aligned}$$

et $\mathcal{P}(n)$ est bien vérifiée.

Démonstration – Suite

Par principe de récurrence forte, $\mathcal{P}(n)$ est vraie $\forall n \in \mathbb{N}$. Cela signifie que l'appel `exp_rapide_rec x n` calculera bien x^n quelque soit le choix des valeurs d'entrée `x` et `n`.

III. 3. 2ème technique : invariant de boucle

La technique de l'**invariant de boucle** permet de montrer la correction d'algorithmes itératifs (présentant des boucles) et/ou modifiant le contenu de variables (variables en C, références en OCaml).

Elle sera à tester en premier dans le cas d'algorithmes présentant des boucles et des instructions d'affectation (paradigme impératif).

Définition 8 (Invariant de boucle)

Étant donné un algorithme et une boucle dans cet algorithme, un **invariant de boucle** est une propriété qui, quelles que soient les entrées valides fournies :

— est valide au début du tout premier tour de boucle

— à chaque tour, si la propriété est vrai au début du tour, alors elle est vraie à la fin du tour de boucle ; la propriété peut être temporairement invalide au milieu du tour de boucle.

Attention. Attention, dans les boucles `for`, l'opération d'incrémentation du compteur de boucle est souvent cachée : en fait, l'incrémentation $i \leftarrow i + 1$ est la toute dernière opération effectuée à la fin de chaque tour.

Remarque. Le plus difficile est de trouver ces invariants. Le plus souvent, les invariants de boucle font référence aux variables et données modifiées par la boucle elle-même. Dans sa forme la plus simple, un invariant de boucle peut être une relation arithmétique entre les valeurs de différentes variables du programme.

Méthodologie 2.
Pour les preuves de préservation d'un invariant de boucle, il est nécessaire d'être rigoureux et de **donner des noms distincts aux valeurs des variables en début de tour et en fin de tour**. Par convention, nous noterons avec un symbole prime (par exemple a') les valeurs des variables en fin de tour.

III. 3. a. Correction pour l'exponentiation naïve itérative

```
# let exp_naive x n =

  assert (n >= 0);

  let r = ref 1.0 in
  let i = ref 0 in
  while (!i < n )
  do
    r := !r *. x;
    i := !i + 1
  done;
  !r;;
val exp_naive : float -> int -> float = <fun>
# exp_naive 2.0 (-1);;
Exception: Assert_failure ("//toplevel//", 60
# exp_naive 2.0 3;;
- : float = 8.
# exp_naive (-1.5) 4;;
- : float = 5.0625
```

Démonstration

On pose comme invariant la propriété :

$$\mathcal{P} : r = x^i ,$$

où r, x et i désignent respectivement le contenu des variables (mutables) `r`, `x` et `i` du code ci-dessus.

Initialisation : Juste avant le premier tour de boucle, on a $r = 1$ et $i = 0$ donc $\mathcal{P}(0)$ est vraie.

Préservation : Imaginons que l'on a effectué i tours de boucle. On suppose que la propriété a été conservée lors des tours de boucles précédents, et donc que $\mathcal{P}$ est vraie au démarrage du tour suivant.

On se place justement au début de ce tour suivant. On note r, x et i le contenu de ces variables au début de ce tour, et r', x' et i' leur contenu en fin de tour.

— `x` n'est pas modifiée dans la boucle donc $x' = x$.

— L'opération d'affectation met à jour `r` :

$$r' = r \times x = x^i \times x = x^{i+1}$$

— `i` est ensuite incrémenté donc $i' = i + 1$

Au final, on a bien $r' = x^{i+1} = x^{i'} = x'^{i'}$ ce qui montre que la propriété est vraie à la fin du tour de boucle

Ainsi la propriété $\mathcal{P}$ est vraie à la fin de chaque tour de boucle et en particulier à la fin du dernier tour qui advient pour $i = n$, on a donc $r = x^n$ à la fin du code. Comme r est la valeur retournée, l'algorithme répond bien à la spécification : il est correct.

III. 3. b. Preuve de correction pour l'exponentiation rapide itérative

```
# let rec exp_rapide x n =

  let r = ref 1.0 in
  let k = ref n in
  let a = ref x in

  while (!k > 0)
  do
    if (!k mod 2 = 1) then
      r := !r *. !a;
    a := !a *. !a;

    k := !k / 2
  done;
  !r;;
val exp_rapide : float -> int -> float = <fun>
# exp_rapide 2.0 (-1);;
- : float = 1.
# exp_rapide 2.0 3;;
- : float = 8.
# exp_rapide (-1.5) 4;;
- : float = 5.0625
```

On rappelle la vision itérative de cet algorithme :

$$x^n = \underbrace{x^{n \bmod 2}}_{r} \times (\underbrace{x^2}_{a})^k$$

$$x^n = \underbrace{x^{n \bmod 2} \times (x^2)^{k \bmod 2}}_{r} \times (\underbrace{x^4}_{a})^{\lfloor \frac{k}{2} \rfloor}$$

Et on peut à nouveau, en posant $k' = \lfloor \frac{k}{2} \rfloor$, faire la même opération :

$$x^n = \underbrace{x^{n \bmod 2} \times (x^2)^{k \bmod 2} \times (x^4)^{k' \bmod 2}}_{r} \times (\underbrace{x^8}_{a})^{\lfloor \frac{k'}{2} \rfloor} \ldots \text{etc}$$

Démonstration

On pose comme invariant la propriété :

$$\mathcal{P} : r \times a^k = x^n \,,$$

où r, k et a désignent respectivement le contenu des variables (mutables) `r`, `k` et `a` du code ci-dessus. x et n sont les valeurs d'entrée et restent constantes dans la fonction.

Initialisation : Avant le 1er tour de boucle, on a $k = n$, $a = x$ et $r = 1$ donc $\mathcal{P}(0)$ est vraie.

Préservation : Imaginons que l'on a effectué i tours de boucle. On suppose que la propriété a été conservée lors des tours de boucles précédents et donc que $\mathcal{P}$ est vraie au début du tour suivant. On note r, k et a le contenu de ces variables au début de ce tour, et r', k' et a' leur contenu en fin de tour. Comme $\mathcal{P}$ est vraie en début de tour par hypothèse, on a :

$$r \times a^k = x^n$$

Si k est impair $k = 2p + 1$ **:** alors l'instruction conditionnelle est exécutée et à la fin du tour, on a : $r' = r \times a$, $a' = a^2$ et $k' = p$. Dans ce cas :

$$r' \times a'^{k'} = r \times a \times (a^2)^p = r \times a^{1+2p} = r \times a^k = x^n$$

et la propriété est donc toujours vérifiée en fin de tour car $r' \times a'^{k'} = x^n$.

Si k est pair $k = 2p$ **:** alors l'instruction conditionnelle n'est pas exécutée et à la fin du tour, on a : $r' = r$, $a' = a^2$ et $k' = p$. Dans ce cas :

$$r' \times a'^{k'} = r \times (a^2)^p = r \times a^{2p} = r \times a^k = x^n$$

et la propriété est donc toujours vérifiée en fin de tour car $r' \times a'^{k'} = x^n$.

Dans les deux cas, la propriété est vraie en fin de tour.

Ainsi la propriété $\mathcal{P}$ est vraie à la fin de chaque tour de boucle et en particulier à la fin du dernier tour qui advient pour $k = 0$. En particulier, lorsque la boucle se termine, on a donc $r \times a^0 = x^n$ c'est-à-dire $r = x^n$.
Comme r est la valeur retournée, l'algorithme répond bien à la spécification : il est correct.

Remarque. Les preuves utilisant des invariants de boucle ne sont pas des preuves par récurrence au sens strict.

IV. Preuves pour l'algorithme de recherche dichotomique

IV. 1. Terminaison de l'algorithme de recherche par dichotomie

On rappelle l'algorithme de recherche par dichotomie ici implémenté de manière itérative :

```c
int bin_search(int valeur, int *tab_trie, unsigned int n)
{

  assert(tab_trie != NULL);

  unsigned int l =  0;
  unsigned int r = n-1;
  unsigned int m;
  int val_mid;

  while (r >= l)
  {
    m = ( r + l )/2; // c'est uen division euclidienne
    val_mid = tab_trie[m];

    if (val_mid == valeur) // valeur trouvée!
      return m;


    if (valeur < val_mid) // la valeur recherchée est donc dans la partie gauche du tableau
      r = m - 1;
    else // sinon elle est dans la partie droite
      l = m + 1;
  }

  // si on arrive ici, c'est qu'on n'a pas trouvé la valeur recherchée
  return -1;
}
```

On note $t = [t_0; \ldots; t_{n-1}]$ le tableau de taille $n > 0$ et v la valeur que l'on recherche dans le tableau.
Plusieurs suites peuvent être introduites pour l'étude de cet algorithme. Tout d'abord les suites $(r_k)_{k\in\mathbb{N}}$ et $(l_k)_{k\in\mathbb{N}}$ qui représentent l'évolution des deux indices droite (r comme *right*) et gauche (l comme *left*) de la fenêtre de recherche. On note m_k l'indice central de la fenêtre de recherche, qui évolue donc lui aussi.
On a donc un système de trois suites couplées :

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

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

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

L'indice k correspond au nombre d'itérations effectuées dans la boucle `while` de l'algorithme.
On introduit la suite :

$$u_k = r_k - l_k + 1$$

Montrons que cette suite est bien un variant.
Cette suite est bien à valeurs entières car r_k et l_k sont des indices, donc des entiers (on peut le montrer de manière évidente par récurrence).

$$u_0 = (n-1) - 0 + 1 = n > 0$$

Montrons que $(u_n)_{n\in\mathbb{N}}$ est décroissante.

$$\begin{aligned}
u_{k+1} - u_k &= r_{k+1} - l_{k+1} + 1 - (r_k - l_k + 1) \\
&= (r_{k+1} - r_k) - (l_{k+1} - l_k) \\
&= \begin{cases} m_k - 1 - r_k & \text{si } v < t_{m_k} \\ 0 & \text{si } v > t_{m_k} \end{cases} - \begin{cases} 0 & \text{si } v < t_{m_k} \\ m_k + 1 - l_k & \text{si } v > t_{m_k} \end{cases} \qquad (1) \\
&= \begin{cases} m_k - r_k - 1 & \text{si } v < t_{m_k} \\ (l_k - 1) - m_k & \text{si } v > t_{m_k} \end{cases}
\end{aligned}$$

Montrons par récurrence :

$$\mathcal{P}(k) : \; l_k - 1 \leq m_k \leq r_k + 1$$

Cas de base : $m_0 = \lfloor \frac{0 + (n-1)}{2} \rfloor$ vérifie bien la propriété $-1 \leq m_0 \leq n$, et $\mathcal{P}(0)$ est vraie.

Hérédité : Supposons $\mathcal{P}(k)$ vraie.

Si $v < t_{m_k}$ **:** Dans ce cas, nous avons :

$$l_{k+1} = l_k \,,$$

$$r_{k+1} = m_k - 1$$

et par hypothèse de récurrence

$$l_k - 1 \leq m_k \leq r_k + 1 \,.$$

Majorons à droite en utilisant les propriétés de la partie entière et l'hypothèse de récurrence :

$$\begin{aligned}
m_{k+1} &= \lfloor \frac{l_{k+1} + r_{k+1}}{2} \rfloor \\
&< \frac{l_{k+1} + r_{k+1}}{2} + 1 = \frac{l_k + r_{k+1}}{2} + 1 \\
&\leq \frac{m_k + 1 + r_{k+1}}{2} + 1 \text{ car } l_k \leq m_k + 1 \text{ par hypothèse de récurrence} \\
&= \frac{m_k - 1 + 2 + r_{k+1}}{2} + 1 \text{ car } r_{k+1} = m_k - 1 \\
&= \frac{r_{k+1} + 2 + r_{k+1}}{2} + 1 \\
&\leq r_{k+1} + 1 + 1 = r_{k+1} + 2
\end{aligned}$$

Minorons à gauche en utilisant les propriétés de la partie entière et l'hypothèse de récurrence :

$$\begin{aligned} m_{k+1} &= \lfloor \frac{l_{k+1} + r_{k+1}}{2} \rfloor \\ &\geq \frac{l_{k+1} + r_{k+1}}{2} \text{ en vertu de la proposition 1} \\ &= \frac{l_k + m_k - 1}{2} \text{ car } l_k = l_{k+1} \text{ et } r_{k+1} = m_k - 1 \\ &\geq \frac{l_k + l_k - 2}{2} \text{ car } m_k - 1 \geq l_k - 2 \text{ par hypothèse de récurrence} \\ &\geq l_k - 1 = l_{k+1} - 1 \text{ car } l_k = l_{k+1} \end{aligned}$$

Si $v > t_{m_k}$ **:**

$$l_{k+1} = m_k + 1$$
$$r_{k+1} = r_k$$

et par hypothèse de récurrence

$$l_k - 1 \leq m_k \leq r_k + 1$$

Majorons à droite en utilisant les propriétés de la partie entière et l'hypothèse de récurrence :

$$\begin{aligned} m_{k+1} &= \lfloor \frac{l_{k+1} + r_{k+1}}{2} \rfloor \\ &< \frac{l_{k+1} + r_{k+1}}{2} + 1 \text{ en vertu de la définition 1} \\ &= \frac{m_k + 1 + r_k}{2} + 1 \text{ car } l_{k+1} = m_k + 1 \texttt{ et } r_{k+1} = r_k \\ &\leq \frac{(r_k + 1) + 1 + r_k}{2} + 1 \text{ car } m_k \leq r_k + 1 \text{ par hypothèse de récurrence} \\ &= r_{k+1} + 2 \end{aligned}$$

Minorons à gauche en utilisant les propriétés de la partie entière et l'hypothèse de récurrence :

$$\begin{aligned} m_{k+1} &= \lfloor \frac{l_{k+1} + r_{k+1}}{2} \rfloor \\ &\geq \frac{l_{k+1} + r_k}{2} \text{ en vertu de la définition 1} \\ &\geq \frac{l_{k+1} + (m_k - 1)}{2} \text{ car } r_k \geq m_k - 1 \text{ par hypothèse de récurrence} \\ &= \frac{l_{k+1} + l_{k+1} - 2}{2} = l_{k+1} - 1 \text{ car } m_k = l_{k+1} - 1 \end{aligned}$$

Dans les deux cas, on a donc réussi à montrer

$$l_{k+1} - 1 \leq m_{k+1} < r_{k+1} + 2$$

et comme m_k est entier par définition de la partie entière, on a en fait :

$$l_{k+1} - 1 \leq m_{k+1} \leq r_{k+1} + 1$$

Ainsi $\mathcal{P}(k+1)$ est vraie.

Revenons maintenant à notre expression :

$$u_{k+1} - u_k = \begin{cases} m_k - r_k - 1 & \text{si } v < t_{m_k} \\ (l_k - 1) - m_k & \text{si } v > t_{m_k} \end{cases}$$

Avec la propriété que nous venons de démontrer, $u_{k+1} - u_k \leq 0$ et donc u_k est décroissante.

Ainsi, soit la suite $(u_k)_{k\in\mathbb{N}}$ devient négative à partir d'une certaine itération k_0, soit elle est stationnaire à partir d'un certain rang.

Nous allons montrer qu'en fait, même si u_k est stationnaire, elle l'est sur une valeur inférieure ou égale à 0.
Supposons que la suite soit stationnaire à partir d'un certain rang :

$$\exists k_s,\ \forall k \geq k_s,\ u_{k+1} = u_k \Leftrightarrow \exists k_s \forall k \geq k_s,\ \begin{cases} r_k + 1 = m_k & \text{si } v < t_{m_k} \\ l_k - 1 = m_k & \text{si } v > t_{m_k} \end{cases}$$

Si $v < t_{m_k}$ **:**

$$\frac{r_k + l_k}{2} - 1 < r_k + 1 \leq \frac{r_k + l_k}{2} \text{ en vertu de 1}$$

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

$$\Leftrightarrow \quad r_k + l_k - 4 < 2r_k \leq r_k + l_k - 2$$

$$\Leftrightarrow \quad l_k - 4 < r_k \leq l_k - 2$$

$$\Leftrightarrow \quad l_k - 3 \leq r_k \leq l_k - 2$$

$$\Leftrightarrow \quad r_k = l_k - 2 \text{ ou } r_k = l_k - 3$$

Si $r_k = l_k - 2$, alors

$$u_k = r_k - l_k + 1 = l_k - 2 - l_k + 1 = -1 < 0$$

Si $r_k = l_k - 3$ alors

$$u_k = l_k - 3 - l_k + 1 = -2 < 0$$

Si $v > t_{m_k}$ **:**

$$\frac{r_k + l_k}{2} - 1 < l_k - 1 \leq \frac{r_k + l_k}{2} \text{ en vertu de } 1$$

$$\Leftrightarrow \quad \frac{r_k + l_k}{2} \leq l_k < \frac{r_k + l_k}{2} + 1$$

$$\Leftrightarrow \quad r_k + l_k < 2l_k \leq r_k + l_k + 2$$

$$\Leftrightarrow \quad r_k < l_k \leq r_k + 2$$

$$\Leftrightarrow \quad r_k + 1 \leq l_k \leq r_k + 2$$

$$\Leftrightarrow \quad l_k = r_k + 1 \text{ ou } l_k = r_k + 2$$

Si $l_k = r_k + 1$ alors

$$u_k = r_k - r_k - 1 + 1 = 0$$

Si $l_k = r_k + 2$, alors

$$u_k = r_k - l_k + 1 = r_k - r_k - 2 + 1 = -1 < 0$$

Finalement, dans tous les cas la suite $(u_k)_{\mathbb{N}}$ devient inférieure ou égale à 0 à partir d'une certaine itération k_0. Au niveau algorithmique, cela signifie qu'il existe une itération k_0 pour laquelle :

$$u_{k_0} = r_{k_0} - l_{k_0} + 1 \leq 0 \Leftrightarrow r_{k_0} - l_{k_0} \leq -1 < 0$$

Ainsi, à l'itération k_0 de la boucle `while`, le test de boucle devient faux et la boucle cesse. L'algorithme se termine donc.

Remarque. La preuve de la version récursive de cet algorithme est tout à fait analogue, en utilisant le même variant.

IV. 2. Preuve de correction de l'algorithme de recherche dichotomique récursif

```
# let recherche_dicho_rec valeur tab_trie =
  let rec aux v tab_trie l r =
    match (l, r) with
    | (l, r) when r < l -> -1
    | _ -> let m = (l+r)/2 in
           match tab_trie.(m) with
           | val_mid when val_mid = v -> m
           | val_mid when v < val_mid -> aux v tab_trie l (m-1)
           | _ -> aux v tab_trie (m+1) r
  in
  let n = Array.length tab_trie in
  aux valeur tab_trie 0 (n-1);;
val recherche_dicho_rec : 'a -> 'a array -> int = <fun>
# let tab_trie2 = tri_a_bulles [| -5; 2; 3; 7; -1; 53; 42; -8|];;
val tab_trie2 : int array = [|-8; -5; -1; 2; 3; 7; 42; 53|]
# recherche_dicho_rec 7 tab_trie2;;
- : int = 5
# recherche_dicho_rec 11 tab_trie2;;
- : int = -1
```

Soit un tableau $t = [t_0; t_1; \ldots; t_{N-1}]$ de taille $N \in \mathbb{N}$ **trié par ordre croissant**.

On note $\mathcal{P}(n)$ la propriété : « Pour toute paire (l, r) tels que $n = r - l + 1 \geq 0$ et $n \leq N$, et pour toute valeur v :

- si v est présente dans le sous-tableau $[t_l; \ldots t_r]$ alors l'appel `aux v t l r` renvoie alors l'indice d'une occurrence de cette valeur.
- si v n'est pas présente dans le sous-tableau $[t_l; \ldots; t_r]$ alors `aux v t l r` renvoie la valeur -1

»

2

Nous allons utiliser une récurrence forte pour montrer la correction de cet algorithme.
Cas de base : $\mathcal{P}(0)$ est vraie car dans ce cas, r est strictement inférieur l : $[t_l; \ldots t_r]$ est un tableau vide, la valeur v ne peut donc être présente dans ce tableau. L'algorithme ne rentre jamais dans la boucle, dont le test est faux, et répond bien -1 dans ce cas, conformément à la propriété.
Hérédité : Pour $n \geq 1$, $l \leq r$ et on a $l \leq m \leq r$ car

$$l = \frac{l+l}{2} \leq \frac{l+r}{2} < \lfloor \frac{l+r}{2} \rfloor + 1 \leq \frac{l+r}{2} + 1 \leq \frac{r+r}{2} + 1 = r + 1$$

donc

$$l < m + 1 \leq r + 1$$

en ajoutant -1 dans chaque membre de l'encadrement et en se rappelant que l et m sont des entiers :

$$l \leq m \leq r$$

Soit $n \geq 1$ et supposons que $\mathcal{P}(k)$ est vraie pour tout $0 < k < n$.
On cherche à montrer $\mathcal{P}(n)$ (c'est-à-dire que l'algorithme renvoie le bon résultat pour un tableau de taille n)

Soit m l'indice central du tableau trié $t = [t_l; t_1; \ldots; t_r]$. On a donc en vertu de la remarque juste au dessus :

$$0 \leq l \leq m \leq r \leq n - 1$$

.

Si $v = t_m$ **:** l'algorithme renvoie l'indice m ce qui est cohérent avec le premier point de la proposition $\mathcal{P}(n)$

Si $v < t_m$ **:** l'algorithme appelle `aux v t l (m-1)`. Pour cet appel $k = (m-1) - l + 1 = m - l \geq 0$ est bien tel que $0 \leq k < n = r - l + 1$ on peut donc appliquer l'hypothèse de récurrence forte.

- Si v est dans le tableau, comme le tableau est trié, v se trouve dans le sous-tableau, lui-aussi trié, $[t_l; \ldots; t_{m-1}]$. L'hypothèse de récurrence nous permet alors d'affirmer que `aux v t l (m-1)` renverra bien l'indice d'une occurrence de cette valeur

2. Lors de la preuve de terminaison, on a montré que, à tout moment de l'algorithme, on a : $l - 1 \leq m \leq r + 1$. On est donc certains que $n \geq -1$. En fait, le cas $n = -1$ est très particulier et correspond au cas de sortie de boucle en l'absence de valeur trouvée.

— Si v n'est pas dans le tableau, il n'est pas non plus présent dans le sous-tableau $[t_l; \ldots; t_{m-1}]$. L'hypothèse de récurrence nous permet alors d'affirmer que `aux v t l (m-1)` renverra -1

Dans les deux cas, l'hypothèse $\mathcal{P}(n)$ est vérifiée

Si $v > t_m$ **:** l'algorithme appelle `aux v t (m+1) r`. Pour cet appel $k = r-(m+1)+1 = r-m$ est bien tel que $0 \leq k < n = r-l+1$ on peut donc appliquer l'hypothèse de récurrence forte.

— Si v est dans le tableau, comme le tableau est trié , v se trouve dans le sous-tableau, lui-aussi trié, $[t_{m+1}; \ldots; t_r]$. L'hypothèse de récurrence nous permet alors d'affirmer que `aux v t (m+1) r` renverra bien l'indice d'une occurrence de cette valeur

— Si v n'est pas dans le tableau, il n'est pas non plus présent dans le sous-tableau $[t_{m+1}; \ldots; t_r]$. L'hypothèse de récurrence nous permet alors d'affirmer que `aux v t (m+1) r` renverra -1

Dans les deux cas, l'hypothèse $\mathcal{P}(n)$ est vérifiée

On a montré que, dans tous les cas, $\mathcal{P}(n)$ est vérifiée. Par récurrence forte, $\mathcal{P}(n)$ est vraie pour tout tableau de taille n. En particulier, elle est vraie pour $n = N$ et valide donc cet algorithme de recherche pour l'ensemble du tableau.

V. Complexité

V. 1. Complexité temporelle

V. 1. a. Définition

Définition 9 (Complexité temporelle)

> On appelle complexité temporelle d'un algorithme l'évolution du coût de cet algorithme (en terme de temps de calcul ou en terme de nombre d'opérations), en fonction de l'évolution de ses paramètres d'entrée.

La complexité temporelle peut être évaluée :

dans le pire des cas : c'est-à-dire dans le cas où les entrées amènent à la situation la plus défavorable en terme de coût ;

dans le meilleur des cas : c'est-à-dire dans le cas où les entrées amènent à la situation la plus favorable en terme de coût ;

en moyenne : c'est-à-dire en faisant la moyenne des complexités pour toutes les entrées possibles d'une même taille ; Cette définition est simple à manipuler lorsque le domaine des entrées de taille n est fini. Dans le cas d'une nombre infini de possibilités pour les entrées de taille n, il faudra procéder à une modélisation probabiliste ;

en terme de coût amorti : c'est-à-dire en lissant la complexité sur une séquence d'invocations successives et dépendantes de l'algorithme.

On peut donner des ordres de grandeur de complexité temporelle en faisant :

une étude empirique : en mesurant le temps de calcul effectif sur des jeux de valeurs d'entrée nombreux, variés et représentatifs, comme lorsque l'on effectue une étude statistique avec échantillonnage ;

une étude théorique : on estime alors le temps de calcul en comptant le nombre d'opérations effectuées par l'algorithme en fonction des entrées fournies.

V. 1. b. Que compte-ton ?

Dans le cas de l'étude théorique, il semblerait nécessaire de préciser les modalités du comptage des opérations. Le temps d'exécution d'un programme dépend de l'ensemble des opérations effectuées. Cependant, les différentes opérations élémentaires (opérations arithmétiques, tests, accès mémoire...) n'ont pas toutes le même coût. Doit-on compter uniquement les opérations arithmétiques ? Ou également les instructions d'affectation, de test ? Donne-t-on le même poids à toutes les opérations, ce qui n'est pas du tout représentatif de la réalité matérielle ?

En réalité, ce degré de détail n'est généralement pas significatif car on s'intéresse non pas au nombre exact d'opérations effectuées, mais plutôt à la manière dont le temps de calcul va évoluer lorsque le volume d'information à traiter va augmenter. On s'intéresse au **comportement asymptotique** de la complexité lorsque les paramètres dont dépendent cette complexité vont grandir.

Ainsi, faire un décompte exact de l'ensemble des opérations toutes catégories confondues, en plus d'être difficile, n'aurait guère de sens : on additionnerait des choses qui ne sont pas

toujours comparables. Raisonner en termes d'ordres de grandeur est par nature imprécis, puisque l'on ne donne que des **profils d'accroissement** sans rien dire de la complexité effective pour une taille donnée, mais cela reste souvent l'énoncé le plus honnête que l'on puisse formuler sans hypothèse supplémentaire sur les machines utilisées pour exécuter le programme.

Remarque. Dans ce cadre, la notion de complexité ne correspond pas à la difficulté ressentie à concevoir un algorithme. Un programme peut être très facile à écrire, mais avoir un temps d'exécution très lent, alors qu'un autre programme beaucoup plus sophistiqué pourra être nettement plus efficace.

V. 2. Outils pour l'étude théorique de complexité

V. 2. a. Calcul de sommes

Nous terminons cette première présentation en rappelant quelques résultats de calcul de sommes, très classiques en mathématiques, qui doivent être connus. Elle doivent également pouvoir être redémontrées rapidement (cf vos cours de mathématiques).

Expression	Somme	Valeur	Equivalent	Ordre
$1+2+3+\ldots+n$	$\sum_{k=0}^{n} k$	$\frac{n\,(n+1)}{2}$	$\sim \frac{n^2}{2}$	$O(n^2)$
$1+4+9+\ldots+n^2$	$\sum_{k=0}^{n} k^2$	$\frac{n\,(n+1)\,(2n+1)}{6}$	$\sim \frac{n^3}{3}$	$O(n^3)$
$1+2+4+\ldots+2^n$	$\sum_{k=0}^{n} 2^k$	$2^{n+1}-1$	$\sim 2^{n+1}$	$O(2^n)$
$1+\frac{1}{2}+\frac{1}{3}+\ldots+\frac{1}{n}$	$\sum_{k=0}^{n} \frac{1}{k}$	H_n	$\sim \ln(n)$	$O(\log_2(n))$
$1+\frac{1}{2}+\frac{1}{4}+\ldots+\frac{1}{2^n}$	$\sum_{k=0}^{n} \frac{1}{2^k}$	$2-\frac{1}{2^n}$	~ 2	$O(1)$
$\log_2(1)+\log_2(2)+\log_2(3)+\ldots+\log_2(n)$	$\sum_{k=0}^{n} \log_2(k)$	$\log_2(n!)$	$\sim n\log_2(n)$	$O(n\log_2(n))$

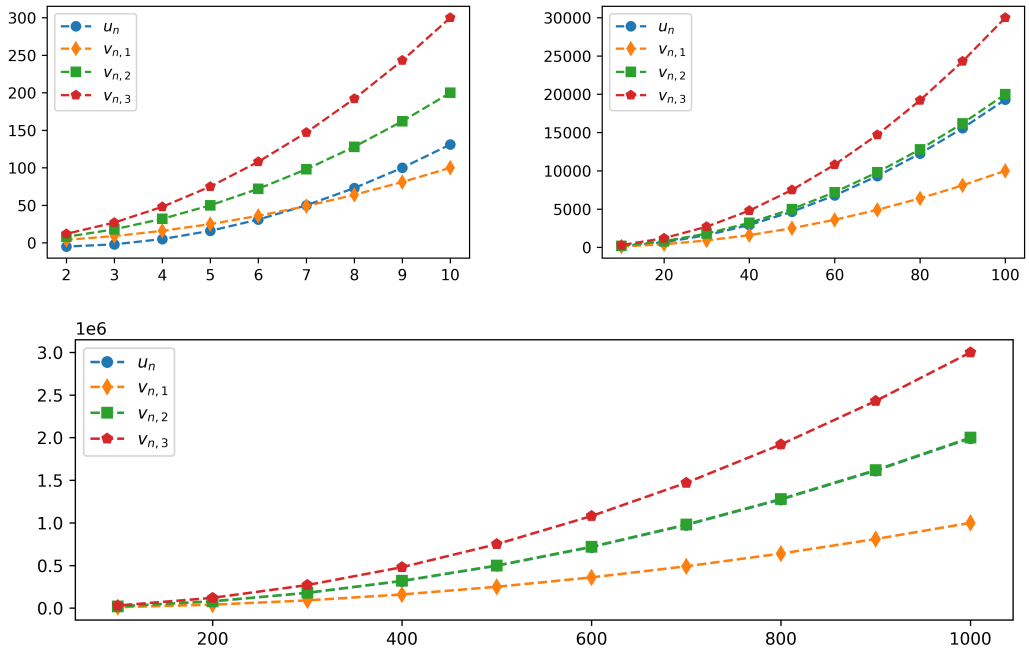
V. 2. b. Notations de Landau

La complexité ne présente véritablement d'intérêt pour analyser les performances d'un algorithme que pour de grandes valeurs de la taille n des données. Ce **comportement asymptotique** suffit, d'une part pour répartir les algorithmes dans des catégories de complexités, d'autre part pour les comparer entre eux. On s'attache donc à déterminer

l'**ordre de grandeur** des complexités plus que leur expression exacte[3] pour de *grandes* valeurs de n. Par exemple, pour une complexité est de la forme $2n^2 - 7n + 1$, le terme en n^2 joue un rôle prédominant dès que n dépasse une certaine valeur. Est-il alors encore nécessaire de conserver le reste de l'information pour caractériser l'algorithme ?
Pour répondre à cette question, introduisons quatre suites qui pourraient être associées aux complexités de programmes traitant des données de taille n, entier naturel non nul.

$$\forall n \in \mathbb{N}^* \qquad u_n = 2n^2 - 7n + 1 \qquad v_{n,1} = n^2 \qquad v_{n,2} = 2n^2 \qquad v_{n,3} = 3n^2$$

Les graphiques suivants présentent les évolutions de ces suites dans des intervalles de valeurs entières de plus en plus larges. Les pointillés ne sont ajoutés que pour montrer les enveloppes des courbes des fonctions sous-jacentes à l'évolution de ces suites.



Sur le premier graphique, les évolutions de chaque suite sont distinctes. Pour les premières valeurs de n jusqu'à 7, la suite (u_n) prend des valeurs inférieures à celles des autres suites avant de devenir supérieure à $(v_{n,1})$. Pour des valeurs intermédiaires de n, de quelques dizaines d'unités à la centaine, les évolutions de (u_n) et de $(v_{n,2})$ sont encadrées par celle de $(v_{n,1})$ inférieurement et de $(v_{n,3})$ supérieurement. Mais leurs évolutions semblent se rapprocher. Sur le troisième graphique, ce rapprochement est tel que les graphes de (u_n) et de $(v_{n,2})$ se superposent pratiquement. L'encadrement par les suites $(v_{n,1})$ et $(v_{n,3})$ est conservé. Ces observations mènent à exprimer l'idée qu'asymptotiquement, les suites (u_n) et de $(v_{n,2})$ présentent des comportements similaire : le cours de mathématiques définit rigoureusement cette observation en terme de suites asymptotiquement équivalentes. Elles appartiennent à une même catégorie de suites en terme de comportement asymptotique, celle des suites qui se comportent comme la plus simple d'entre elles : la suite (n^2). On dit que les suites (u_n) et de $(v_{n,2})$ appartiennent à la même **classe de complexité** notée $O(n^2)$. Cette expression se lit : *grand O de n au carré*
Cette notation, appelée notation de Landau, vient du mot allemand *Ordnung* qui signifie *de l'ordre de*.
Pour établir ces ordres de grandeur, on utilisera donc généralement la notation de Landau O appliquée pour des **fonctions à variable(s) entière(s)** (il peut y avoir plusieurs

3. Cette dernière n'est d'ailleurs pas accessible tant sont nombreux les facteurs, parfois difficiles à identifier, qui la définissent.

variables) que l'on étudie de manière asymptotique, c'est-à-dire lorsque cette ou ces variable(s) entière(s) tendent vers $+\infty$
On se place pour simplifier dans le cas où la complexité a été exprimée en fonction d'une seule variable n.

Définition 10 (Notation « grand O » O)

Soit $f : \mathbb{N} \to \mathbb{R}$ et $g : \mathbb{N} \to \mathbb{R}$ deux fonctions.
On dit que g est « un grand O de f » en $+\infty$ si :

$$\exists k \in \mathbb{R}^+, \exists n_0 \in \mathbb{N}, \forall n \geq n_0,\ g(n) \leq k\, f(n)$$

On note $g(n) \in \underset{n \to +\infty}{O}(f(n))$

Exemple 2

Reprenons l'exemple des suites ci-dessus. Cette définition permet donc d'écrire $(u_n) \in O((v_{n,1}))$ ou même $(v_{n,2}) \in O((v_{n,1}))$. En pratique, cette écriture est remplacée par une notation, abusive mais consacrée par l'usage, où le signe d'appartenance est remplacé par le signe égal et les suites sont assimilées à leurs expressions. Ainsi, on écrit [a] :

$$u_n = O(n^2) \qquad v_{n,2} = O(n^2)$$

Cette définition permet même d'écrire :

$$v_{n,1} = O(n^2) \qquad v_{n,3} = O(n^2)$$

En d'autres termes, les quatre suites ont le même comportement asymptotique. Quand il s'agira d'algorithmes, nous dirons que les algorithmes appartiennent à la même **classe de complexité asymptotique**

a. Nous adoptons cette écriture dans toute la suite du cours.

Remarque. Il est évident que $n^2 = O(n^2)$. Mais on peut aussi noter que $n^2 = O(n^3)$, $n^2 = O(n^4)$ et ainsi de suite. Mais pour caractériser une complexité par un grand O, on cherche généralement celui qui est le plus proche de la complexité analysée. Ainsi, nous écrivons préférentiellement :

$$2n^2 - 7n + 1 = O(n^2)$$

Ajoutons également que la notation grand O ne compare pas les nombres d'opérations, mais, à des constantes près, la vitesse de croissance des fonctions qui comptent les nombres d'opérations. Les constantes importent peu, ce qui permet d'écrire $v_{n,2} = O(n^2)$ ou $v_{n,3} = O(n^2)$ malgré la présence des coefficients 2 et 3.

Définition 11 (Notation équivalent $\sim$)

Soit $f : \mathbb{N} \to \mathbb{R}$ et $g : \mathbb{N} \to \mathbb{R}$ deux fonctions.
On dit que g est équivalente à f en $+\infty$ si :

$$\lim_{n\to+\infty} \frac{g(n)}{f(n)} = 1$$

On note $g(n) \underset{n\to+\infty}{\sim} f(n)$

Exemple 3

Considérons la fonction $g(n) = \frac{3n^2}{2} + 2n + 5$. On a l'équivalence :

$$g(n) \underset{n\to+\infty}{\sim} \frac{3n^2}{2}$$

On a donc notamment $g(n) \in \underset{n\to+\infty}{O}(n^2)$. Mais on a également des bornes asymptotiques moins précises : $g(n) \in \underset{n\to+\infty}{O}(n^3)$, $g(n) \in \underset{n\to+\infty}{O}(2^n)$...

Propriété 4

L'ordre de grandeur d'une somme est l'ordre de grandeur de son terme dominant.
L'ordre de grandeur d'un produit est le produit des ordres de grandeur.

Dans la suite, cette notation permet des simplifications comme par exemple :

$$\begin{aligned} n + n^2 &= O(n^2) \\ n + n\log_2 n &= O(n\log_2 n) \\ n^2 + n\log_2 n &= O(n^2) \\ n\sqrt{n} + \log_2 n &= O(n^{3/2}) \\ 2^n + n^{10} &= O(2^n) \end{aligned}$$

Remarque. En général, nous raisonnerons avec des $\underset{n\to+\infty}{O}$. Mais pour obtenir des informations plus précises, on peut raisonner en termes d'équivalence $\underset{n\to+\infty}{\sim}$. Cette notation est plus précise car elle requiert l'explicitation des constantes multiplicatives devant le terme dominant, ce qui peut finalement avoir un impact quantitatif réel sur le coût de l'algorithme et peut jouer dans la comparaison d'algorithmes ayant des comportements asymptotiques du même ordre.
Dans le cas où l'on utilise des équivalents, on se concentre en général sur certaines opérations représentatives de la complexité de l'algorithme. Par exemple, pour les algorithmes agissant sur des tableaux, ce sont souvent les accès mémoires (en lecture ou écriture) qui gouvernent la complexité globale de l'algorithme.

Pour résumé :

— dans la plupart des cas, on se ramènera à un ordre de grandeur global exprimé avec un O. Dans ce cas, on ne cherche donc pas à déterminer la constante multiplicative devant le terme dominant et ainsi, on peut effectuer un calcul grossier en considérant que tout groupe d'opérations élémentaires (arithmétique, test, affectation) compte pour une unité.

— dans certains cas particulier, par exemple si l'on veut estimer avec finesse la complexité pour des données de grandes tailles, ou si l'on souhaite comparer deux algorithmes ayant des complexités du même ordre, on peut raisonner avec des équivalents. Dans ce cas, on ne décompte que certains types d'opérations jugées représentatives.

V. 2. c. Un peu de vocabulaire

Complexité	Nom associé	Cas typique
$\underset{n\to+\infty}{O}(1)$	constante	série finie d'instructions, absence de boucle
$\underset{n\to+\infty}{O}(\log_2(n))$	logarithmique	algorithmes dichotomiques (exp rapide, recherche dicho...)
$\underset{n\to+\infty}{O}(n)$	linéaire	boucle simple, recherche séquentielle
$\underset{n\to+\infty}{O}(n\log_2(n))$	quasi-linéaire	diviser pour régner (semestre 2)
$\underset{n\to+\infty}{O}(n^2)$	quadratique	deux boucles imbriquées, tri par insertion, tri à bulles [4]
$\underset{n\to+\infty}{O}(n^3)$	cubique	trois boucles imbriquées, produit de matrices
$\underset{n\to+\infty}{O}(n^k)$, $k\in\mathbb{N}^*$	polynomiale	k boucles imbriquées
$\underset{n\to+\infty}{O}(2^n)$	exponentielle	recherche exhaustive, problème de satisfiabilité en logique
$\underset{n\to+\infty}{O}(n!)$	factorielle	résolution exhaustive du problème du voyageur de commerce

V. 2. d. Quelques chiffres pour fixer les idées

Les différents profils de complexité donnent des valeurs très différentes lorsque la taille n du problème devient grande.

Complexité	$n=10$	$n=10^2$	$n=10^3$	$n=10^6$	$n=10^9$
$\log_2(n)$	3	7	10	20	30
n	10	10^2	10^3	10^6	10^9
$n\log_2(n)$	30	7×10^2	10^4	2×10^7	3×10^{10}
n^2	10^2	10^4	10^6	10^{12}	10^{18}
2^n	10^3	$>10^{30}$	$>10^{300}$	$>10^{300\,000}$	$>10^{300\,000\,000}$

A titre de comparaison, 2×10^{79} est une estimation du nombre d'atomes dans l'univers observable...

Pour obtenir des ordres de grandeur encore plus parlants, on peut convertir ces valeurs en un temps d'exécution. Des mesures expérimentales en conditions réelles montre qu'un microprocesseur ordinaire actuel réalise de l'ordre du milliard d'opérations flottantes (FLOPS *FLoating-point OPerationS*) par minute (bien sûr, il y a beaucoup plus de cycles d'horloges que d'instructions car certaines instructions prennent plusieurs dizaines de cycles d'horloge) :

4. Une complexité factorielle est encore pire qu'une complexité exponentielle, en vertu de l'équivalent de Stirling $n!\sim\sqrt{2\pi n}e^{-n}n^n$ que vous avez étudié en mathématiques.

Complexité	$n = 10$	$n = 10^2$	$n = 10^3$	$n = 10^6$	$n = 10^9$
$\log_2(n)$	inst.	inst.	inst.	inst.	inst.
n	inst.	inst.	inst.	ms	1 min
$n\log_2(n)$	inst.	inst.	inst.	1 s	30 min
n^2	inst.	inst.	1 s	$>$ 16 h	$>$ 1900 ans
2^n	inst.	$>$ âge de l'univers	$>$ âge de l'univ.	$>$ âge de l'univ.	$>$ âge de l'univ.

V. 3. Exemples : complexité temporelle pour des algorithmes itératifs

V. 3. a. Recherche séquentielle

Illustrons cette notion de complexité en évaluant celle de la fonction de recherche séquentielle d'une valeur `v` dans un tableau d'entiers `t` de taille `n`.

```c
int seq_search(int v, int *t, unsigned int n)
{
  assert(t != NULL);

  unsigned int idx = 0;

  while (idx < n && t[idx] != v)
    idx = idx + 1;

  if (idx == n) // si idx == n, on n'a pas trouvé la valeur
    idx = -1;

  return idx;
}
```

La durée de la recherche est directement liée au nombre de tours effectués dans la boucle `while`.

Complexité dans le meilleur des cas. Supposons que la valeur recherchée soit égale à celle du premier élément du tableau. Dans ce cas, la boucle s'arrête dès la premier tour puisque l'élément est trouvé. La durée d'exécution de la fonction se réduit donc aux instructions liées à son appel, à la déclaration de l'entier `i`, à l'instruction `while (idx<n)`, au test `if (t[idx] ==v)` et enfin au renvoi de la valeur de l'indice `return idx`. Que la taille du tableau soit petite ou grande, si l'élément recherché est en première position dans le tableau, le coût temporel de l'algorithme sera le même. La complexité dans le meilleur des cas est donc *constante*, quelle que soit la taille du tableau.

$$\forall n \in \mathbb{N} \qquad C_{\text{meilleur}}(n) = \text{cste}$$

La constante ne dépend que de la durée des instructions exécutées pour aboutir au résultat.

Complexité dans le pire des cas. Supposons à présent que l'élément recherché soit le dernier élément du tableau. La boucle compare alors tous les éléments d'indice $0, 1, \ldots, n-2$ avec `v`, le test étant à chaque fois négatif. C'est seulement quand idx atteint la valeur $n-1$ que le test est positif et que la fonction renvoie $n-1$ par l'instruction `return idx`. Dans ce cas, en plus des instructions d'entrée dans la fonction, n tests `if (t[idx] == v)` sont effectués, $n-1$ incrémentations `i = i + 1` sont faites et un `return idx` renvoie

la valeur $n-1$. Si on note C_{pire} le coût temporel de la recherche dans cette situation, on peut écrire :

$$C_{\text{pire}}(n) = n \times C_{\text{test}} + (n-1) \times C_{\text{incrémentation}} + cste$$

Là encore, la constante correspond aux coûts annexes, indépendants de n. En réorganisant un peu cette expression, on peut écrire :

$$C_{\text{pire}} = an + b$$

où $a > 0$ et b sont des constantes qui ne dépendent que du coût de certaines instructions. La constante a est positive, ce qui montre que C_{pire} croît avec n. Dans le cas présent, la complexité dans le pire des cas est **linéaire** en la taille du tableau.

V. 3. b. Tri par insertion

```
# let tri_insertion tab =
  let n = Array.length tab in
  for i = 1 to (n-1) do    n-1 tours de boucle
    let v = tab.(i) in
    let j = ref i in
    while ( !j >= 1 && v < tab.((!j-1)) ) do  i tours de boucle au pire
      tab.(!j) <- tab.(!j-1); (* on est pas obligé de faire des swaps
 ici car on décale tout vers la droite. Il suffit de conserver la pre
mière valeur comme nous l'avons fait dans v *)
      j := !j - 1
    done;
    tab.(!j) <- v
  done;
  tab;;
val tri_insertion : 'a array -> 'a array = <fun>
# let tab_trie = tri_insertion_rec [ 6; -5; 7; 1];;
val tab_trie : int list = [-5; 1; 6; 7]
# let tab_trie2 = tri_insertion_rec [ 1; 7; 3; 2; -5; 7; 1];;
val tab_trie2 : int list = [-5; 1; 1; 2; 3; 7; 7]
# let tab_trie3 = tri_insertion_rec [ 1; ];;
val tab_trie3 : int list = [1]
# let tab_trie4 = tri_insertion_rec [ 'f'; 'z'; 'a'; 'c'; 'w'];;
val tab_trie4 : char list = ['a'; 'c'; 'f'; 'w'; 'z']
# let tab_trie5 = tri_insertion_rec [ 1.3; -5.1; 8.1; -3.0; -4.1];;
val tab_trie5 : float list = [-5.1; -4.1; -3.; 1.3; 8.1]
```

Complexité dans le meilleur des cas. Le cas d'un tableau initial déjà trié semble *a priori* correspondre à une situation favorable car un algorithme bien pensé devrait pouvoir tenir compte de cette configuration particulière. Cela signifie qu'un élément déjà bien placé n'a pas besoin d'être inséré où que ce soit. L'algorithme de tri par insertion travaille effectivement ainsi puisque la boucle `while` n'est mise en œuvre que si une insertion est nécessaire. Par conséquent, si le tableau est déjà trié, le coût de cette boucle est constant, à chaque tour de la boucle `for`. Ce qui permet d'exprimer la **complexité au mieux** sous la forme :

$$C_{\text{mieux}}(n) = an + b$$

où $a > 0$ et b sont des constantes qui dépendent des coûts des opérations faites pendant ces tours de boucle. La complexité au mieux est donc **linéaire**.

Complexité dans le pire des cas. Le pire des cas correspond *a priori* à celui où tous les éléments sont en ordre inverse de ce qu'on attend. Chaque insertion d'un élément d'indice i compris entre 1 et $n-1$ revient à décaler tous les éléments du sous-tableau gauche d'un cran vers la droite et à placer l'élément à insérer au tout début du tableau.
Par exemple, si `arr = [|4;3;2;1|]`, l'entier `3` doit être inséré avant le `4`, ce qui coûte un décalage pour mener au tableau intermédiaire `[|3;4;2;1|]`. Puis l'entier `2` doit être inséré avant le `3` à l'aide de deux décalages pour mener au tableau `|[2;3;4;1|]`. Enfin,

le `1` est inséré avant le `2` à l'aide de trois décalages. Ainsi, ce sont $1+2+3=6$ décalages qui sont réalisés pour ordonner le tableau.
Ce résultat se généralise aisément à un tableau de n entiers en ordre inverse. Pour chaque élément d'indice i allant de 1 à $n-1$ (boucle `for`), i décalages sont nécessaires pour l'insérer en tête du tableau. Le nombre total de décalages est alors :

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

Exercice 1.
Dans toutes les questions, n et $m<n$ 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;
}
```

V. 4. Exemples : complexité temporelle pour des algorithmes récursifs

V. 4. a. Exponentiation rapide

On rappelle que 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
```

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.
La complexité temporelle de l'algorithme s'exprime en fonction de l'entier n donné en entrée de l'algorithme comme :

$$\begin{aligned} C(0) &= 0 \\ C(n) &= \begin{cases} 1 + C(\lfloor \frac{n}{2} \rfloor) & \text{si } n \text{ est pair} \\ 2 + C(\lfloor \frac{n}{2} \rfloor) & \text{si } n \text{ est impair} \end{cases} \end{aligned}$$

Cas particulier $n = 2^k$. 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.

Cas général. 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 :

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

Preuve de la complexité dans le cas général. 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 :

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

Démonstration

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 $C(n)$:

$$C(0) \quad = 0$$

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

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 V. 4. a. :

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

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

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 :

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

Par ailleurs, pour tout n tel que $2^k \leq n < 2^{k+1}$ où $k \in \mathbb{N}$, on a

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

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

$$\log_2(n) + 2 \leq C(n) \leq 2\left(\log_2(n) + 1\right)$$

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 $\log_2(n)$. [5]

V. 4. b. Suite de Fibonacci

On considère le code récursif [6] de calcul d'un terme de la suite de Fibonacci $(F_n)_{n\in\mathbb{N}}$.

```c
#include <stdio.h>
#include <assert.h>
#include <stdlib.h>

int fibonacci(int n)
{
  assert(n >= 0);

  if (n == 0)
    return 0;
  if (n == 1)
    return 1;

  return fibonacci(n-1) + fibonacci(n-2);
}
```

Analysons la complexité temporelle et notons C_n le nombre d'additions nécessaires au calcul de F_n.
Le code ci-dessus permet d'établir $C_0 = 0$, $C_1 = 0$.
Ensuite, pour tout entier naturel n, $C_{n+2} = C_{n+1} + C_n + 1$. En effet, pour calculer F_{n+1}, C_{n+1} additions sont nécessaires ; pour calculer F_n, il en faut C_n ; pour calculer C_{n+2} il faut ajouter F_{n+1}, F_n et ensuite faite une addition supplémentaire entre ces deux termes, d'où le $+1$.
On peut, en utilisant une formulation matricielle et la théorie des valeurs propres, trouver une expression explicite pour la suite C_n :

$$\forall n \in \mathbb{N} \qquad C_n = \frac{1}{2\varphi - 1}\left(\varphi^{n+1} - (1-\varphi)^{n+1}\right) - 1$$

où $\varphi = (1+\sqrt{5})/2$ est le nombre d'or.
Finalement, $C_n = O(\varphi^n)$. La complexité temporelle est donc exponentielle !

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

V. 5. Calcul de complexité moyenne

Définition 12 (Complexité moyenne)

La complexité temporelle moyenne d'un algorithme correspond au coût temporel moyen obtenu en considérant l'ensemble de toutes les entrées possibles de l'algorithme.
Il s'agit d'un point de vue mathématique d'un calcul d'**espérance de la variable aléatoire** qui donne, pour chaque entrée possible, le coût temporel associé.

V. 5. a. Cas où le nombre d'entrées possibles est fini

Dans ce cas, on peut utiliser des techniques de **dénombrement** pour expliciter le coût temporel de chaque entrée et ainsi calculer le coût moyen.
Pour illustrer ce cas, on considère la fonction `premier_indice_vrai` suivante :

```c
#include <stdbool.h>
#include <stdio.h>
#include <assert.h>

int premier_indice_vrai(bool *tab, int n)
{

  assert(tab != NULL);
  assert(n >= 0);

  int i = 0;

  while ( i < n && tab[i] != true)
    i ++;

  if (i == n)
    return -1;

  return i;
}
```

La complexité temporelle est mesurée ici en nombre d'accès en lecture aux cases du tableau (c'est un choix arbitraire, mais qui n'a que peu d'importance finalement, nous l'avons vu).

Pour étudier la complexité temporelle, on s'intéresse au coût de l'algorithme pour une taille de tableau fixée n. Cette fonction prend en entrée un tableau de **booléens** : chaque case ne peut prendre que deux valeurs différentes : `true` ou `false`. On a donc 2 choix pour chaque case du tableau.
Au total, on a donc 2^n tableaux possibles en entrée de la fonction pour une taille fixée n : le nombre d'entrées possibles est fini.

On appelle Ω_n l'ensemble de tous les tableaux booléens de taille n qui peuvent être donnés en entrée de l'algorithme. Cet ensemble est appelé, en probabilités, l'univers des possibles : ici, Ω_n est un ensemble fini et son cardinal (nombre d'éléments) est :

$$\operatorname{Card} \Omega_n = 2^n$$

Étudions maintenant le coût (nombre de cases lues) associé à chacun de ces tableaux. On propose deux méthodes d'analyse.

Méthode 1 : dénombrement par complexité. Cette méthode consiste à rassembler et à compter tous les tableaux qui vont aboutir au même nombre de lecture de cases. Le nombre de lecture de cases est forcément un entier, et varie de 1 (il faut au moins une lecture de case) à n (on balaie toutes les cases du tableau).
On appelle T_k l'ensemble des tableaux qui génèrent k lectures de cases pour cet algorithme, avec $1 \leq k < n$. Tous les tableaux de T_k sont tels que :

— les cases d'indice entre 0 et $(k-2)$ contiennent la valeur `false` ;

— la case d'indice $(k-1)$ contient la valeur `true` ;

— les $n-1-k+1 = n-k$ cases suivantes peuvent prendre au choix `true` ou `false`.

Il y a donc $n-k$ cases pour lesquelles j'ai le choix entre 2 valeurs et donc il y a 2^{n-k+1} tableaux dans T_k :

$$\operatorname{Card} T_k = 2^{n-k},\ \forall k \in [\![1,\ n-1]\!]$$

T_n est un cas un peu particulier : la formule ci-dessous ne marche pas pour $k = n$ car il n'y a pas un mais deux tableaux (les pires !) qui amènent à effectuer n lectures de cases :

— le tableau ne contenant que des `false` (cas où l'algorithme renvoie -1) ;

— le tableau ne contenant que des `false` sauf la dernière valeur `true`.

$$\operatorname{Card} T_n = 2$$

Formalisons maintenant le calcul de complexité en terme de probabilités.
On considère la variable aléatoire X ci-dessous qui donne la complexité associée à chaque tableau possible :

$$\begin{array}{lll} X_n : \ \Omega_n & \to & [\![1,\ n]\!] \\ \quad t \text{ (tableau de taille } n & \to & X_n(t) \text{ nombre de lectures de cases pour le tableau } t \end{array}$$

Calculer la complexité moyenne de l'algorithme pour une entrée de taille n revient à calculer l'espérance mathématique de cette variable aléatoire :

$$C_{\text{moy}}(n) = E[X_n] = \sum_{k=1}^{n} (k \times \mathbb{P}(X_n = k))$$

Il nous reste à évaluer la probabilité $\mathbb{P}(X = k)$. Pour cela, on effectue une modélisation probabiliste. En l'absence d'informations supplémentaires sur l'origine et/ou la nature des données, **on fait généralement l'hypothèse que tous les tableaux d'entrée sont équiprobables**. Cela nous permet d'écrire :

$$\mathbb{P}(X = k) = \frac{\operatorname{Card} T_k}{\operatorname{Card} \Omega_n}$$

On a donc :

$$\mathbb{P}(X = k) = \frac{2^{n-k}}{2^n} = \frac{1}{2^k},\ \forall 1 \leq k < n\,,$$
$$\mathbb{P}(X = n) = \frac{2}{2^n} = \frac{1}{2^{n-1}}$$

On peut donc finaliser notre calcul :

$$\begin{aligned} C_{\text{moy}}(n) &= \sum_{k=1}^{n} k\,\mathbb{P}(X = k) \\ &= \sum_{k=1}^{n-1} \left(k \times \frac{1}{2^k} \right) + n \times \frac{1}{2^{n-1}} \\ &= \sum_{k=1}^{n-1} \left(\frac{k}{2^k} \right) + \frac{n}{2^{n-1}} \end{aligned}$$

Cette somme n'est pas triviale à calculer. On peut cependant s'en sortir grâce à une **méthode de téléscopage** :

$$\begin{aligned} C_{\text{moy}}(q+1) - C_{\text{moy}}(q) &= \sum_{k=1}^{(q+1)-1} \left(\frac{k}{2^k}\right) + \frac{q+1}{2^q} \\ &\quad - \left(\sum_{k=1}^{q-1} \left(\frac{k}{2^k}\right) + \frac{q}{2^{q-1}}\right) \\ &= \frac{q}{2^q} + \frac{q+1}{2^q} - \frac{q}{2^{q-1}} \\ &= \frac{q+q+1-2q}{2^q} = \frac{1}{2^q} \end{aligned}$$

On a toujours, par télescopage (compensations) :

$$C_{\text{moy}}(n) - C_{\text{moy}}(0) = \sum_{q=0}^{n-1} \left(C_{\text{moy}}(q+1) - C_{\text{moy}}(q)\right)$$

$C_{\text{moy}}(0) = 0$ car pour un tableau vide, on ne lit aucune case, et on peut poursuivre le calcul (somme géométrique) :

$$C_{\text{moy}}(n) = \sum_{q=0}^{n-1} \frac{1}{2^q} = \frac{1 - \left(\frac{1}{2}\right)^n}{1 - \frac{1}{2}} = 2 - \frac{1}{2^{n-1}}$$

Nous avons donc montré :

$$C_{\text{moy}}(n) = 2 - \frac{1}{2^{n-1}}$$

Méthode 2 : dénombrement par case lue. Dans cette méthode, on se fixe une case d'indice j, on dénombre tous les tableaux qui vont amener la lecture de cette case, quelque soit la valeur stockée dans cette case. On appelle $T'_{j,n}$ l'ensemble des tableaux à n cases qui vont engendrer la lecture de la case d'indice j, $0 \leq j \leq n-1$.

— Si $j = 0$, les 2^n tableaux possibles vont engendrer la lecture de cette case :

$$\text{Card}\, T'_0 = 2^n$$

— Si $j = 1$, tous les tableaux vont amener la lecture de cette case sauf les tableaux commençant par `true` : il reste donc 2^{n-1} tableaux qui vont engendrer la lecture de la case d'indice $j = 2$ (pas de choix pour la 1ère case mais 2 choix pour les $n-1$ cases suivantes). Ainsi :

$$\text{Card}\, T'_{1,n} = 2^{n-1}$$

— Pour une case d'indice $j < n-1$ quelconque, $0 \leq j \leq n-2$, les $j-1$ cases précédentes doivent être à `false` et mais nous sommes libres et avons 2 choix pour les $n-1-j+1 = n-j$ cases restantes. Il y a donc 2^{n-j} tableaux qui vont engendrer la lecture de la case j :

$$\text{Card}\, T' j, n = 2^{n-j}$$

— Pour $j = n-1$ (dernière case), seuls 2 tableaux (les pires) vont engendrer la lecture de cette case : le tableau ayant toutes ses cases à `false` et le tableau ayant toutes ses cases à `false` sauf la dernière.

$$\operatorname{Card} T'_{n-1,n} = 2 = 2^{n-(n-1)}$$

On a donc une formule générale qui marche pour tous les cas :

$$\operatorname{Card} T'_{j,n} = 2^{n-j},\ \forall 0 \leq j \leq n-1$$

Cette fois-ci, on considère la variable aléatoire Y_n suivante :

$$\begin{array}{rccl} Y_{j,n} : & \Omega_n & \to & [\![0,\,1]\!] \\ & t & \to & Y_{j,n}(t) = 1 \text{ si la case } j \text{ est lue, 0 sinon} \end{array}$$

Comme la variable aléatoire $Y_{j,n}$ ne peut prendre que deux valeurs, l'espérance n'a que deux termes :

$$\begin{aligned} E[Y_{j,n}] &= 0 \times \mathbb{P}(Y_{j,n} = 0) + 1 \times \mathbb{P}(Y_{j,n} = 1) \\ &= \mathbb{P}(Y_{j,n} = 1) \end{aligned}$$

On doit ici faire une hypothèse probabiliste pour continuer à avancer. On suppose donc que tous les tableaux sont équiprobables : parmi les 2^n tableaux de l'univers des possibles, j'ai autant de chance de tomber sur un tableau que sur un autre. Cela permet d'écrire que :

$$\mathbb{P}(Y_{j,n} = 1) = \frac{\operatorname{Card} T' j, n}{\operatorname{Card} \Omega_n} = \frac{2^{n-j}}{2^n} = \frac{1}{2^j}$$

On a donc :

$$E[Y_{j,n}] = \frac{1}{2^j}$$

Si l'on revient à la définition de $Y_{j,n}$, on s'aperçoit que la complexité, qui correspond ici au nombre de cases lues, est associée à la variable aléatoire :

$$X_n = \sum_{j=0}^{n-1} Y_{j,n}$$

On a, par linéarité de l'espérance mathématique :

$$C_{\text{moy}}(n) = E[X_n] = \sum_{j=0}^{n-1} E[Y_{j,n}] = \sum_{j=0}^{n-1} \frac{1}{2^j}$$

Mais cette fois-ci, on a immédiatement la somme des termes d'une suite géométrique de raison $\dfrac{1}{2}$, que l'on sait calculer :

$$C_{\text{moy}}(n) = \frac{1 - \left(\dfrac{1}{2}\right)^n}{1 - \dfrac{1}{2}} = 2 - \frac{1}{2^{n-1}}$$

Et on retrouve le même résultat, ce qui est très rassurant !

Analyse de ce résultat. Nous avons montré de deux manières possibles que :

$$C_{\text{moy}}(n) = 2 - \frac{1}{2^{n-1}}$$

On remarque déjà que cette complexité est bien comprise entre 1 (meilleur des cas, on ne lit qu'une case) et n (pire des cas, on lit toutes les cases).

Mais on observe aussi que $C_{\text{moy}}(n) \in O(1)$, la complexité temporelle est donc en moyenne... constante ! On a même mieux : pour des très grands tableaux, on lira en moyenne 1 ou 2 cases avant de tomber sur une valeur vraie. Cela peut paraître surprenant, mais c'est logique : en fait, j'ai une chance sur deux de m'arrêter dès la première case !

V. 5. b. Cas où le nombre d'entrées possibles est infini

Dans la plupart des cas, le nombre d'entrées possible est infini. C'est le cas, par exemple, pour un algorithme de tri d'un tableau d'entiers : il y a une infinité de choix d'entiers pour chaque case du tableau d'entrée.

Dans ce cas, on ne peut pas dénombrer les tableaux engendrant telle ou telle chose, mais on peut par contre reprendre l'idée des classes d'équivalence qui a été utilisée ci-dessus. Il s'agit alors de regrouper les tableaux engendrant le même comportement en terme de complexité dans une même famille, appelée classe d'équivalence. On pourra ensuite **redéfinir** les variables aléatoires non pas sur un ensemble d'entrées infini, mais sur un nombre fini de classes d'équivalences. Le calcul de l'espérance mathématique en sera alors facilité.

Pour illustrer cette approche, nous allons étudier la complexité moyenne de l'algorithme de tri par insertion.

Complexité de l'insertion de l'élément t_i. On ne s'intéresse dans un premier temps qu'à la sous-boucle `while`. On considère donc un indice i fixé : on s'intéresse au coût temporel de l'insertion de l'élément t_i dans le sous-tableau gauche trié $[t_0, t_1, \ldots, t_{i-1}]$.
L'infinité de sous-tableaux triés de taille i constitue l'univers des possibles Ω_i de ce sous-problème. Ω_i est donc, contrairement au cas précédent, un ensemble infini : on ne peut pas parler de cardinal !
On choisit (arbitrairement !) de définir le coût temporel comme le nombre de tests à réaliser pour pouvoir insérer t_i au bon endroit dans le sous-tableau gauche. Ce coût dépend du nombre d'éléments de ce sous-tableaux qui sont plus grands strictement que t_i.

- S'il n'y a aucun élément plus grand que t_i, il faut quand même un test de comparaison avec t_{i-1} pour s'en rendre compte, il y a donc 1 test ;
- Si seul l'élément t_{i-1} est plus grand que t_i, alors il faut 2 tests ;
- S'il y a $k < i$ éléments (t_{i-k} jusqu'à t_{i-1}) plus grands que t_i, alors il faut $(k+1)$ tests
- Si tous les éléments du sous-tableau gauche sont plus grands que t_i, alors on a fait i tests pour s'en rendre compte et insérer t_i tout à gauche du sous-tableau.

On voit donc que l'infinité des sous-tableaux $[t_0, t_1, \ldots, t_{i-1}]$ peuvent être rangés par classe d'équivalence.
Pour $k \in [\![0, i]\!]$, on note $T_{k,i}$ la classe d'équivalence contenant l'infinité de tableaux triés de taille i tels que les k derniers éléments sont plus grands que t_i.
On a :

$$\Omega_i = \cup_{k=0}^{i} T_{k,i}$$

On note X_i la variable aléatoire :

$$\begin{array}{rlcl} X_i : & \Omega_i & \to & [\![0,\,1]\!] \\ & t = [t_0,\,t_1,\,\ldots,\,t_{i-1}] \text{ trié} & \to & X_i(t) \text{ nombre de tests pour insérer } t_i \text{ dans } t \end{array}$$

La complexité moyenne du sous-problème d'insertion de t_i se calcule comme l'espérance mathématique de X_i

$$C_{\text{moy}}(i) = E[X_i]$$

Cependant, il est bien difficile de calculer cette espérance : il faudrait faire une moyenne sur un nombre infini de sous-tableaux possibles... une intégrale sur des espaces de tableaux... On peut pourtant s'en sortir en remarquant que, comme tous les tableaux $t \in T_{k,i}$ engendrent le même nombre de tests, on peut redéfinir X_i en une fonction $\tilde{X}_i$ définie sur les classes d'équivalences :

$$\begin{array}{rlcl} \tilde{X}_i : & \cup_{k=0}^{i} T_{k,i} & \to & [\![0,\,1]\!] \\ & t \in T_{k,i} & \to & \tilde{X}_i(t) = k+1 \end{array}$$

On peut alors faire une hypothèse probabiliste non pas sur les sous-tableaux possibles, mais sur les classes d'équivalences : on suppose que toutes les classes d'équivalence sont équiprobables. Autrement dit, si je pioche un sous-tableau de taille i au hasard parmi l'infinité des tableaux possibles, j'ai autant de chance qu'il s'agisse d'un tableau de la classe $T_{0,i}$ (aucun élément plus grand que t_i) ou bien de la classe $T_{1,i}$ (une élément plus grand que t_i) ou de n'importe laquelle des i classes d'équivalence $T_{k,i}$, $k \in [\![0,\,i]\!]$.
Comme il y a $i+1$ classes d'équivalence équiprobables, on a donc :

$$\mathbb{P}(t \in T_{k,i}) = \mathbb{P}(\tilde{X}_i = k+1) = \frac{1}{i+1}$$

On peut alors facilement calculer l'espérance de X_i en effectuant cette fois-ci une somme **finie** sur les classes d'équivalence

$$\begin{aligned} C_{\text{moy}}(i) &= E[X_i] = \sum_{k=0}^{i-1} (k+1)\mathbb{P}(\tilde{X}_i = k+1) + \frac{i}{i+1} \\ &= \frac{1}{i+1}\sum_{k=0}^{i-1}(k+1) + \frac{i}{i+1} \\ &= \frac{i(i+1)}{2(i+1)} + \frac{i}{i+1} \\ &= \frac{i}{2} + \frac{i}{i+1} \end{aligned}$$

Retour à la complexité globale. Par propriété de linéarité de l'espérance mathématique, il suffit de sommer le coût de l'insertion de tous les éléments t_i :

$$C_{\text{moy}}(n) = \sum_{i=0}^{n-1} C_{\text{moy}}(i) = \sum_{i=0}^{n-1}\left(\frac{i}{2} + \frac{i}{i+1}\right) = \frac{1}{2}\sum_{i=0}^{n-1} i + \sum_{i=0}^{n-1}\frac{i}{i+1}$$

La première somme est une somme classique facile à calculer :

$$\frac{1}{2}\sum_{i=0}^{n-1} i = \frac{(n-1)\,n}{4} \in O(n^2)$$

La seconde somme n'est pas triviale, mais on peut très facilement la majorer par n car $\frac{i}{i+1} \leq 1$.
Ainsi $\sum_{i=0}^{n-1} \frac{i}{i+1} \in O(n)$, ce qui signifie que ce terme n'a pas d'impact d'un point de vue asymptotique.
Finalement, la complexité moyenne du tri par insertion est :

$$C_{\text{moy}}(n) \sim \frac{n^2}{4} \in O(n^2)$$

Pour résumer l'analyse de la complexité de l'algorithme de tri par insertion, on a :

- complexité au mieux : linéaire en la taille du tableau ;
- complexité au pire : $\sim \frac{n^2}{2}$ quadratique en la taille n du tableau ;
- complexité moyenne : $\sim \frac{n^2}{4}$ quadratique en la taille n du tableau.

V. 6. Complexité spatiale

Définition 13 (Complexité spatiale)

On appelle complexité spatiale d'un algorithme le coût d'un algorithme en terme d'utilisation des ressources mémoire (mémoire de masse ou mémoire vive), en fonction des entrées.
Lors de l'évaluation de la complexité spatiale, **on ne comptabilise que les entités mémoire créées en plus pour le travail de l'algorithme (variables de travail). On ne comptabilise pas le coût mémoire lié au stockage des données d'entrée**.

Tout ce qui a été dit pour la complexité temporelle s'applique également pour la complexité spatiale :

- elle peut être évaluée dans le pire des cas, le meilleur des cas, en moyenne ou en terme de coût amorti ;
- elle peut être étudiée de manière théorique ou empirique ;
- c'est surtout le comportement asymptotique de cette complexité qui va nous intéresser.

Attention. Pour bien évaluer la complexité spatiale, il faut prendre en compte :

- les allocations dynamiques sur le tas (`malloc`)
- les allocations semi-automatiques sur la **pile**, et en particulier l'espace mémoire occupé par l'empilement des blocs d'activation dans la pile, notamment si la récursivité n'est pas terminale (cf Séquence 6)

Exemple 4

Analysons la complexité spatiale de la fonction `fibonacci` étudiée plus haut : dans cette fonction, chaque appel récursif mène à un empilement sur la pile d'exécution : cette dernière voit donc sa taille augmenter jusqu'à ce que l'argument de la fonction `fibo` vale `0` ou `1` (cas de base). Mais attention : certes, la récursivité n'est pas terminale et on empile les appels, mais lorsque l'on atteint le cas de base, on dépile également. Si l'on dessine l'arbre des appels récursifs comme nous l'avons fait pour les tours de Hanoï (faites-le, c'est un excellent exercice, ou aller revoir la Séquence 6), on voit bien que l'on va effectuer un parcours en profondeur de l'arbre : c'est-à-dire que la complexité spatiale sera linéaire.

Le nombre d'appels total réalisés (le nombre de nœuds de l'arbre) est certes exponentiel (en $O(2^n)$), mais comme on dépile aussi régulièrement, le nombre de blocs maximal empilés à un instant donné ne dépassera jamais n (la profondeur de l'arbre) pour le calcul du terme de rang n.

Ainsi, la complexité spatiale de la fonction `fibonacci` est linéaire.

Remarque. En OCaml, le complexité temporelle domine toujours la complexité spatiale car toute variable doit être initialisée. Il y a donc au moins autant de zones mémoires allouées que d'opérations élémentaires (type affectation).

Remarque. En pratique, on s'intéresse souvent à l'utilisation optimale des ressources mémoires et on s'inquiète surtout des **pics d'utilisation de la mémoire**, qui peuvent faire déborder la pile ou le tas, voire mener à des phénomènes de *swapping* lorsque l'espace disponible physiquement en mémoire vive n'est plus suffisant et qu'il faut aller chercher de l'espace de travail dans le disque dur. D'où l'importance de :

- rester sobre dans l'allocation mémoire ;
- veiller à éliminer toutes les fuites mémoires (perte de l'adresse permettant d'accéder à une zone mémoire allouée manuellement sur le tas) ;
- bien désallouer l'espace mémoire alloué manuellement sur le tas lorsqu'il n'est plus nécessaire (`free`).

En OCaml, l'utilisation des ressources est automatiquement optimisée par l'interpréteur.

Remarque (Hors-Programme). L'interpréteur OCaml alloue beaucoup de mémoire dans le tas. La plupart des variables locales aux fonctions sont stockées dans le tas au lieu d'être stockées dans la pile comme cela est normalement prévu. Les blocs d'activation de la pile ne contiennent alors que des pointeurs vers ces variables locales. Ce mécanisme automatique permet de simuler le renvoie de plusieurs valeurs par une fonction et de s'affranchir de l'implémentation manuelle du passage par adresse par exemple.