Fusion

12 janvier 2026

On s'intéresse aujourd'hui à la fusion de deux tableaux triés, un algorithme des plus classiques. On prend en entrée deux tranches s_1 et s_2 triées d'éléments de type T générique (implémentant *Copy*) et on produit en sortie un vecteur trié contenant tous les éléments de s_1 et s_2.

s_1	1	3	4	8		s_2	2	5	9	10

fusion	1	2	3	4	5	8	9	10

1 Algorithme classique

1. À l'aide de deux indices i_1 et i_2 écrire un algorithme de fusion.

Solution:

```
pub fn fusion_vec<T: Ord + Copy>(s1: &[T], s2: &[T]) -> Vec<T> {
    // on va faire la fusion de papi et mamie
    let n = s1.len() + s2.len();
    let mut i1 = 0; // avec deux indices pour savoir ou on est
    let mut i2 = 0;
    let mut fusion = Vec::with_capacity(n);
    loop {
        match (s1.get(i1), s2.get(i2)) {
            (Some(t1), Some(t2)) => {
                if t1 < t2 {
                    fusion.push(*t1);
                    i1 += 1;
                } else {
                    fusion.push(*t2);
                    i2 += 1;
                }
            }
            (None, Some(_)) => {
                fusion.extend_from_slice(&s2[i2..]);
                break;
            }
            (Some(_), None) => {
                fusion.extend_from_slice(&s1[i1..]);
                break;
            }
            (None, None) => break,
        }
    }
```

```
    fusion
}
```

2. Quelle optimisation peut-on réaliser en fin d'algorithme ?

Solution: Lorsqu'on atteint la fin d'une tranche on peut simplement terminer en copiant le reste de l'autre tranche. On évite d'une part toutes les comparaisons restantes et de plus la copie d'un bloc sera réalisée par un *memcopy* qui est très rapide.

2 Itérateurs

On cherche maintenant à écrire un itérateur réalisant la fusion de deux itérables triés. Il n'est donc plus possible d'utiliser des indices pour savoir où l'on en est. À la place on va utiliser un itérateur peekable. Les itérateurs disposent d'une méthode *peekable* renvoyant un `Peekable<Self>`. Celui-ci est lui-même un itérateur mais dispose également d'une méthode `fn peek(&mut self) -> Option<&<I as Iterator>::Item>` permettant d'examiner l'élément suivant avant l'appel à *next*.

2.1 from_fn

1. À l'aide de `from_fn` implémenter l'algorithme de fusion. On perd malheureusement la possibilité de réaliser l'optimisation de fin d'algorithme.

Solution:

```rust
pub fn fusion<T: Ord, I: IntoIterator<Item = T>, J: IntoIterator<Item = T>>(
    i: I,
    j: J,
) -> impl Iterator<Item = T> {
    // on va faire la fusion du futur
    let mut iter_i = i.into_iter().peekable();
    let mut iter_j = j.into_iter().peekable();
    // le from_fn est l'arme ultime pour eviter d'ecrire un iterateur manuellement
    std::iter::from_fn(move || match (iter_i.peek(), iter_j.peek()) {
        (Some(ti), Some(tj)) => {
            if ti < tj {
                iter_i.next()
            } else {
                iter_j.next()
            }
        }
        (None, Some(_)) => iter_j.next(),
        (Some(_), None) => iter_i.next(),
        (None, None) => None,
    })
}
```

2.2 Implémentation manuelle

Dans un dernier temps on se propose d'implémenter manuellement un type *Fusionneur*. En implémentant le trait *Iterator* on va implémenter *next* mais également *fold* et cette implémentation de *fold* va nous permettre de récupérer une partie de l'optimisation initiale.

1. Implémenter *next*

Solution:

```rust
fn next(&mut self) -> Option<Self::Item> {
    match (self.iter_i.peek(), self.iter_j.peek()) {
        (None, None) => None,
        (None, Some(_)) => self.iter_j.next(),
        (Some(_), None) => self.iter_i.next(),
        (Some(ti), Some(tj)) => {
            if ti < tj {
                self.iter_i.next()
            } else {
                self.iter_j.next()
            }
        }
    }
}
```

2. Implémenter *fold*

Solution:

```rust
fn fold<B, F>(mut self, init: B, mut f: F) -> B
where
    Self: Sized,
    F: FnMut(B, Self::Item) -> B,
{
    let mut valeur_courante = init;

    loop {
        match (self.iter_i.peek(), self.iter_j.peek()) {
            // Note : la lib standard triche en specialisant extend
            // malheureusement on ne peut pas le faire pour le moment...
            (Some(ti), Some(tj)) => {
                valeur_courante = f(
                    valeur_courante,
                    if ti < tj {
                        self.iter_i.next().unwrap()
                    } else {
                        self.iter_j.next().unwrap()
                    },
                )
            }
            (Some(_), None) => return self.iter_i.fold(valeur_courante, f),
            (None, Some(_)) => return self.iter_j.fold(valeur_courante, f),
            (None, None) => return valeur_courante,
        }
    }
}
```

3. Quelle autre méthode de l'itérateur pourrait s'avérer utile ?

Solution: Il faut implémenter `size_hint` pour pré-allouer le vecteur destination à la bonne taille lors du collect.

4. Pourquoi ne récupère t'on pas toute l'optimisation ?

Solution: On évite toutes les comparaisons de fin mais il n'y a pas de garantie que toutes les copies soient réalisées par un *memcopy*.

3 En bonus

1. Implémenter *Peekable*.

Solution: L'implémentation de la lib standard :

```
fn peek(&mut self) -> Option<&I::Item> {
    let iter = &mut self.iter;
    self.peeked.get_or_insert_with(|| iter.next()).as_ref()
}
fn next(&mut self) -> Option<I::Item> {
    match self.peeked.take() {
        Some(v) => v,
        None => self.iter.next(),
    }
}
fn fold<Acc, Fold>(self, init: Acc, mut fold: Fold) -> Acc
where
    Fold: FnMut(Acc, Self::Item) -> Acc,
{
    let acc = match self.peeked {
        Some(None) => return init,
        Some(Some(v)) => fold(init, v),
        None => init,
    };
    self.iter.fold(acc, fold)
}
```

2. Quelle optimisation réaliser lors du *fold* ?

Solution: Pas la peine de faire tous les *match* du *next*.