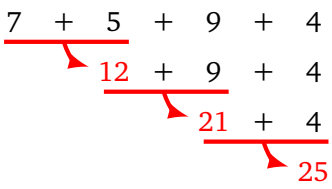
Calculs en parallèle

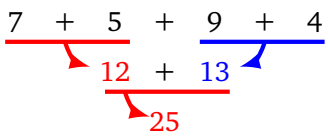
Comment profiter d'avoir plusieurs processeurs (ou plusieurs cœurs dans chaque processeur) pour calculer plus vite ? C'est simple, il suffit de partager les tâches à réaliser afin que tout le monde travaille en même temps, puis de regrouper les résultats. Dans la pratique, ce n'est pas si facile.

Cours 1 (Calculs en parallèle : motivation)**.**
Tu as trouvé un vaccin contre les zombies qui ravagent le monde. Comment distribuer le plus rapidement possible les pilules à 100 personnes, sachant qu'on ne peut être en contact qu'avec une seule personne à la fois ? Méthode séquentielle (un seul processeur) : distribuer les pilules une par une. S'il faut 1 minute pour chaque distribution, il te faudra en tout 100 minutes. Méthode à deux processeurs : donne la moitié des pilules à un camarade (1 minute), puis chacun distribue ses pilules. Au total cela prend environ 50 minutes donc deux fois plus rapide. Peux-tu faire mieux ?

Habituellement on effectue un calcul de façon séquentielle : le calcul est divisé en plusieurs tâches, chacune est exécutée tour à tour pour à la fin donner le résultat.
Dans le calcul en parallèle : on répartit les tâches entre plusieurs processeurs, qui calculent simultanément.
Par exemple : comment calculer $7+5+9+4$? De façon séquentielle on fait $7+5=12$ (1 seconde) puis on fait $12+9=21$ (1 seconde), et enfin $21+4=25$, pour un total de trois secondes. On peut faire mieux avec deux processeurs : le premier calcule $7+5=12$, le second calcule $9+4=13$ (cela fait en tout une seule seconde), puis le premier calcule $12+13=25$. Au total cela a pris deux secondes.



Calculs séquentiels

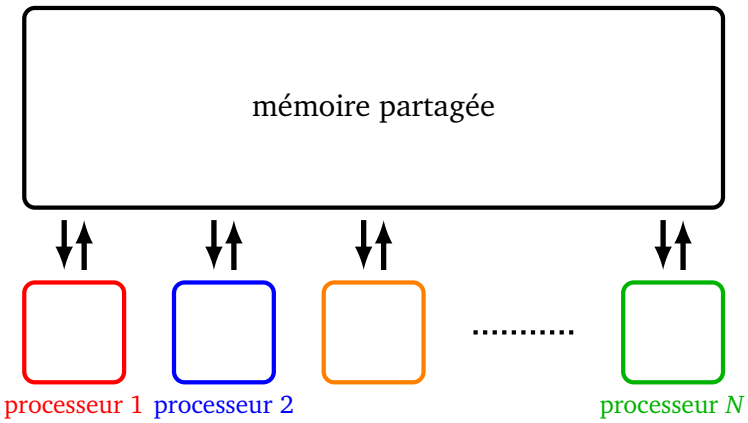


Calculs en parallèle

Attention, ce n'est pas toujours possible d'effectuer les calculs en parallèle : par exemple pour calculer $(7+5)\times 3+2$, il n'y pas d'autre choix que de calculer séquentiellement, car on a besoin des résultats partiels pour continuer les calculs.

Cours 2 (Calculs en parallèle : un modèle)**.**
Modèle.
Voici le modèle d'ordinateur avec lequel nous travaillerons ici :

- une mémoire partagée qui contient les données, stocke les résultats,
- plusieurs processeurs qui fonctionnent simultanément et ont accès à la mémoire.



Nombre de calculs/temps des calculs.
On souhaite mesurer l'efficacité des algorithmes en parallèle. Un algorithme est divisé en calculs élémentaires. Pour un algorithme séquentiel (avec un seul processeur) chaque tâche est exécutée une par une donc le temps des calculs est égal au nombre de calculs à effectuer. Pour un ordinateur qui effectue des calculs en parallèle, on distingue les deux notions :

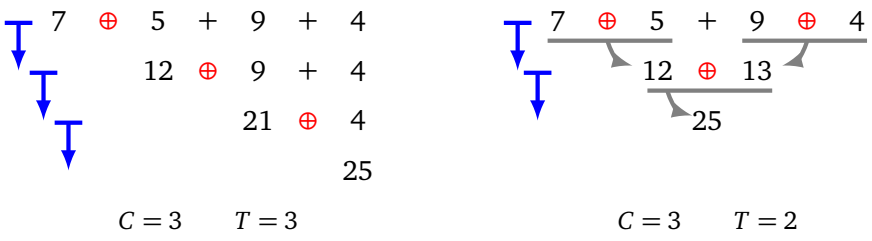
- le ***nombre de calculs*** C est le nombre de tâches élémentaires,
- le ***temps des calculs*** T est le temps total pour effectuer tous les calculs.

Un calcul correspond à une unité de temps. Deux calculs peuvent être faits successivement pour un temps des calculs $T = 2$, mais si les deux calculs sont effectués en parallèle alors $T = 1$.

Exemple 1.
On veut multiplier tous les éléments d'une liste $(x_0, x_1, \dots, x_{n-1})$ par 2, pour obtenir $(2x_0, 2x_1, \dots, 2x_{n-1})$.

- Si le nombre de processeurs est $N = 1$, alors il faut effectuer successivement chacune des opérations $2x_i$, $i = 0, \dots, n-1$. Le nombre de calculs est $C = n$ et le temps des calculs est $T = n$.
- Si on dispose de $N = 2$ processeurs, alors on peut effectuer les calculs $2x_0$ et $2x_1$ en même temps, puis $2x_2$ et $2x_3$. Au final, le nombre total de calculs reste $C = n$, mais par contre le temps des calculs est divisé par 2 : $T = n//2$ (en supposant n pair, ou bien $T = n//2 + 1$ si n est impair).
- Avec $N = 4$ processeurs, on a toujours $C = n$ et $T \simeq n//4$.
- Si on a beaucoup de processeurs ($N \geqslant n$) alors $C = n$ et $T = 1$ car tous les calculs peuvent être effectués en même temps.

Exemple 2.



Reprenons l'exemple vu plus haut du calcul $7 + 5 + 9 + 4$.

- Calculs séquentiels ($N = 1$). Le nombre d'opérations est $C = 3$: c'est le nombre d'additions (notées $\oplus$ en rouge). Le temps des calculs est $T = 3$ (c'est le nombre d'étapes symbolisées par les flèches verticales bleues).

- Calculs en parallèle avec $N = 2$. Le nombre de calculs est toujours $C = 3$ (il y a autant d'additions $\oplus$ effectuées), par contre le temps des calculs est cette fois $T = 2$ car lors de la première étape deux calculs sont effectués en parallèle.

Activité 1 (Modèle et premiers calculs en parallèle)**.**

Objectifs : simuler le travail d'un ordinateur avec plusieurs processeurs calculant en parallèle et programmer nos premiers algorithmes de calculs en parallèle.

On simule le travail d'un ordinateur effectuant des calculs en parallèle. Notre ordinateur reçoit une liste de n instructions sous la forme d'une chaîne de caractères. L'ordinateur possède N processeurs. Alors :

- le nombre de calculs est $C = n$,
- le temps des calculs est $T = n/N$ (arrondi à l'entier supérieur).

Exemple : si les instructions sont `['2+2','3*4','10+1','8-5']` alors l'ordinateur renvoie `[4,12,11,3]`, le nombre de calculs est toujours $C = n = 4$, le temps des calculs dépend du nombre de processeurs. Par exemple avec $N = 2$, le temps des calculs est $T = 2$, mais si $N = 4$ le temps des calculs est $T = 1$.

1. **Notre modèle.**

   Programme une fonction `calcule_en_parallele(liste_instructions,N)` qui reçoit une liste d'instructions (sous la forme de chaînes de caractères). La fonction renvoie d'abord la liste des résultats mais aussi le nombre de calculs C et le temps des calculs T nécessaires aux N processeurs.

   Exemple. Avec les instructions :

   `['2+3','6*7','8-2','5+4','4*3','12//3']`

   et $N = 2$ processeurs la fonction renvoie :

   `[5, 42, 6, 9, 12, 4], 6, 3`

   Le nombre de calculs est $C = 6$, le temps des calculs est $T = 3$. Teste d'autres valeurs de N.

   Indications.

   - `eval(chaine)` permet d'évaluer une expression donnée sous la forme d'une chaîne de caractères. Par exemple `eval('8*3')` renvoie 24.
   - `ceil(x)` (du module `math`) renvoie la partie entière supérieure d'un nombre. Exemple `ceil(3.5)` vaut 4.

2. **Addition de deux vecteurs.**

   On se donne deux vecteurs $\vec{v}_1 = (x_0, x_1, x_2, \ldots, x_{n-1})$ et $\vec{v}_2 = (y_0, y_1, y_2, \ldots, y_{n-1})$ (autrement dit deux listes de nombres). Il s'agit de calculer la somme $\vec{v}_1 + \vec{v}_2 = (x_0 + y_0, x_1 + y_1, \ldots, x_{n-1} + y_{n-1})$.

   Programme une fonction `addition_vecteurs(v1,v2)` qui prend en entrée deux listes de nombres et renvoie le vecteur somme $\vec{v}_1 + \vec{v}_2$.

   Méthode. Cette fonction doit effectuer les calculs en parallèle en utilisant ta fonction `calcule_en_parallele()`. Il y a un calcul à faire pour chaque composante des vecteurs. En plus cette fonction peut renvoyer le nombre de calculs et le temps des calculs.

   Exemple. Avec `v1 = [1, 2, 3, 4]` et `v2 = [10, 11, 12, 13]`, la commande `addition_vecteurs(v1,v2)` renvoie d'une part le résultat `[11, 13, 15, 17]` et si on fixe par exemple le nombre de processeurs à $N = 2$ alors le nombre de calculs est $C = 4$ et le temps des calculs est $T = 2$.

   Indications. Il faut transformer chaque calcul en une instruction sous la forme d'une chaîne de caractères. Par exemple la chaîne obtenue par la commande « `str(x) + '*' + str(y)` » sert pour le calcul « $x \times y$ ».

3. **Somme des termes.**

On se donne un vecteur $\vec{v}$ sous la forme d'une liste de nombres $(x_0, x_1, x_2, \ldots, x_{n-1})$. Il s'agit de calculer la somme :

$$S = x_0 + x_1 + x_2 + \cdots + x_{n-1} = \sum_{i=0}^{n-1} x_i$$

On propose la méthode suivante :

- on calcule en parallèle la somme de deux termes consécutifs :

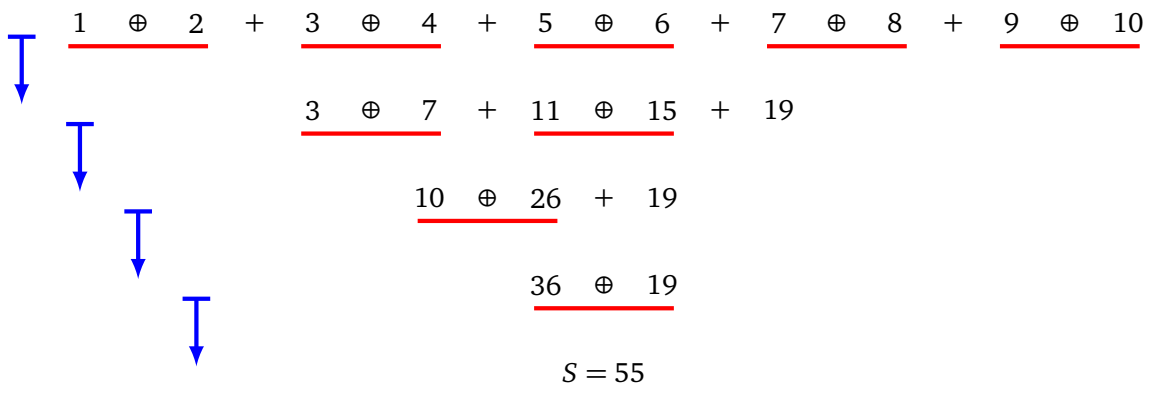
  $$y_0 = x_0 + x_1 \qquad y_1 = x_2 + x_3 \qquad y_2 = x_4 + x_5 \quad \ldots$$

  et de façon générale

  $$y_i = x_{2i} + x_{2i+1}.$$

  On obtient une nouvelle liste $(y_0, y_1, y_2, \ldots)$ deux fois plus courte.
- Puis on recommence avec la liste obtenue : $z_0 = y_0 + y_1$, $z_1 = y_2 + y_3$,... jusqu'à obtenir un seul élément qui est la somme S voulue.

Voici l'algorithme sur l'exemple de la somme $1+2+3+\cdots+10$. Les termes sont regroupés par paires (s'il reste un élément isolé à la fin, il est reporté sur la ligne d'après). Le nombre de calculs est $C = 9$ (le nombre d'opérations $\oplus$ soulignées) et si on dispose de $N = 5$ processeurs alors le temps des calculs est $T = 4$ (le nombre de flèches).



Voici le détail de l'algorithme qui en entrée reçoit une liste $\vec{v} = (x_0, x_1, \ldots, x_{n-1})$:

- Tant que la longueur n de la liste $\vec{v}$ est strictement plus grande que 1 :
  - — Définir une liste $\vec{w}$ dont les composantes y_i sont $y_i = x_{2i} + x_{2i+1}$ pour i variant de 0 à $n//2$.
  - — Si n est impair rajouter à $\vec{w}$ le dernier élément de $\vec{v}$.
  - — Faire $\vec{v} \leftarrow \vec{w}$.
- À la fin la liste est de longueur 1, l'unique terme donne la somme S cherchée.

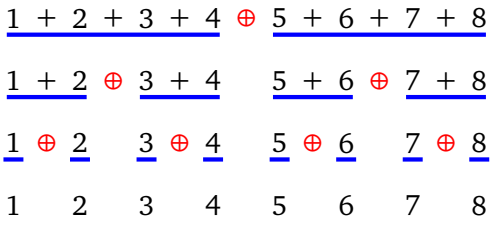
Programme cet algorithme en une fonction `somme(v)`. Cette fonction doit faire appel à la fonction `calcule_en_parallele()` pour le calcul simultané des y_i. Pour cela il faut transformer les opérations $x_{2i} + x_{2i+1}$ en une chaîne de caractères.

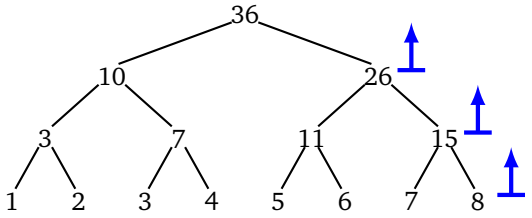
En plus de la somme, programme dans un deuxième temps ta fonction de sorte qu'elle renvoie aussi le nombre total de calculs, ainsi que le temps total des calculs (ce sont les sommes de tous les nombres et temps des calculs intermédiaires).

Exemple. Avec $N = 4$ processeurs et $\vec{v} = (1,2,3,4,5,6,7,8)$ une liste de longueur $n = 8$ alors on trouve une somme de $S = 36$, le nombre de calculs est $C = 7$ (c'est le nombre de signes « + » dans la somme $1+2+3+4+5+6+7+8$) et un temps des calculs de $T = 3$. Avec $N = 2$ processeurs, le temps des calculs devient $T = 4$.

4. **Somme des termes par un algorithme récursif (facultatif).**

L'idée est de séparer notre liste en une partie droite et une partie gauche, puis d'itérer le processus. Voici le schéma sur l'exemple de la somme $1+2+3+\cdots+8$. Sur la figure de gauche, à lire de haut en bas, les divisions successives en partie droite et partie gauche. Sur la figure de droite l'arbre des calculs, à lire depuis le bas vers le haut.





Algorithme.

- — Entête : `somme_recursive(v)`
  — Entrée : $\vec{v} = (x_0, x_1, \ldots, x_{n-1})$ une liste de n nombres.
  — Sortie : la somme S de ses termes
  — Action : fonction récursive.
- **Cas terminaux.**
  — Si $n = 0$, renvoyer 0.
  — Si $n = 1$, renvoyer le seul élément de la liste.
- **Cas général où $n \geqslant 2$.**
  — Séparer la liste $\vec{v}$ en deux sous-listes (à l'aide du rang $n//2$) une sous-liste des termes de gauche `v_gauche` et une sous-liste des termes de droite `v_droite`.
  — Calculer la somme S_g des termes de gauche par l'appel récursif `somme_recursive(v_gauche)`.
  — Calculer la somme S_d des termes de droite par l'appel récursif `somme_recursive(v_droite)`.
  — Renvoyer la somme $S = S_g + S_d$.

Programme cet algorithme en une fonction `somme_recursive(v)`.

Zombies. C'est cette méthode qui s'apparente le plus à la distribution efficace de nos pilules pour contrer les zombies.

Plus difficile. Dans un second temps, modifie ta fonction afin qu'elle renvoie aussi le nombre de calculs et le temps des calculs nécessaires (en supposant qu'il y a suffisamment de processeurs).

5. **Produit scalaire.**

Programme une fonction `multiplication_vecteurs(v1,v2)` calquée sur le modèle `addition_vecteurs()` qui renvoie le produit terme à terme $(x_0 \times y_0, x_1 \times y_1, \ldots, x_{n-1} \times y_{n-1})$. de deux vecteurs $\vec{v}_1 = (x_0, x_1, x_2, \ldots, x_{n-1})$ et $\vec{v}_2 = (y_0, y_1, y_2, \ldots, y_{n-1})$.

À l'aide de la fonction `somme()` déduis-en une fonction `produit_scalaire(v1,v2)` qui calcule le produit scalaire :

$$\langle \vec{v}_1 \mid \vec{v}_2 \rangle = x_0 \times y_0 + x_1 \times y_1 + \cdots + x_{n-1} \times y_{n-1}$$

Calculs. Le produit scalaire nécessite n multiplications, puis $n-1$ additions, donc un total de $C = 2n-1$ calculs. On prend l'exemple de deux vecteurs de longueur $n = 16$. Avec un seul processeur il faudra un temps des calculs $T = 31$. Combien de temps faut-il si on dispose de $N = 8$ processeurs ?

Vérifie expérimentalement que si n est grand, alors on a $T \simeq C/N$.

Activité 2 (Doublons)**.**

Objectifs : retirer les doublons d'une liste à l'aide d'algorithmes adaptés aux calculs en parallèle.

Première méthode : indexation.
Pour cette méthode la liste est une liste d'entiers, par exemple des entiers entre 0 et 99 :

```
liste = [59, 72, 8, 37, 37, 8, 21, 22, 37, 59]
```

On souhaite retirer les doublons, c'est-à-dire obtenir la liste :

```
[59, 72, 8, 37, 21, 22]
```

L'idée est de remplir petit à petit une grande table d'indexation :

- au départ la table contient 100 zéros (un pour chaque entier possible entre 0 et 99),
- pour chaque élément i de la liste initiale on regarde la table au rang i :
  — s'il y a un 0 c'est que l'élément est nouveau, on le conserve dans une nouvelle liste et on place un 1 au rang i de la table,
  — s'il y a déjà un 1 dans la table c'est que l'élément i a déjà été indexé, on ne le conserve pas dans la nouvelle liste.

Programme cet algorithme en une fonction `enlever_tous_doublons(liste)`.

Exemple. Prenons un exemple plus simple avec des entiers de 0 à 9 et la liste :

```
liste = [3, 5, 4, 5, 8, 8, 4]
```

La table d'indexation au départ ne contient que des 0 :

```
table = [0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
```

Puis on parcourt la liste :

- le premier élément est 3 : on place un 1 au rang 3, la table d'indexation est maintenant `[0, 0, 0, 1, 0, 0, 0, 0, 0, 0]` (la numérotation commence au rang 0).
- les éléments suivants sont 5 puis 4, on place des 1 au rang 5 puis au rang 4, la table est maintenant `[0, 0, 0, 1, 1, 1, 0, 0, 0, 0]`,
- ensuite on retrouve l'élément 5, on sait qu'on a déjà pris en compte cet élément car au rang 5 de la table il y a un 1, on ne fait donc rien,
- ensuite l'élément est 8, la table devient `[0, 0, 0, 1, 1, 1, 0, 0, 1, 0]`, puis on retrouve 8 qui ne change rien,
- enfin on trouve 4 que l'on a déjà rencontré et qui ne change pas la table.

On ne retient que les éléments lors de leur première apparition, la liste sans doublons est donc :

```
[3, 5, 4, 8]
```

Calculs en parallèle. Cet algorithme est facile à implémenter en parallèle : la table est située dans la mémoire globale et le parcours des éléments se fait en parallèle, s'il y a un 0 dans la table on conserve l'élément, s'il y a un déjà 1 on l'oublie.

Inconvénients. La méthode présente deux gros inconvénients : d'une part elle n'est valable que pour des listes d'entiers et surtout il faut construire une table qui peut être immense comparée à la liste initiale, par exemple même pour une petite liste d'entiers entre 0 et 999 999 il faut commencer par construire une table de longueur un million. La seconde méthode va remédier à ces deux problèmes.

Seconde méthode : table de hachage.
On souhaite enlever les doublons de la liste :

```
['LAPIN','CHAT','ZEBRE','CHAT','CHIEN',
  'TORTUE','CHIEN','SINGE','SINGE','CHAT']
```

1. **Fonction de hachage.** À une chaîne de caractères et un entier p, on va associer un entier h avec $0 \leqslant h < p$; h sera appelé le *hachage* du mot modulo p. La méthode est la suivante :
   - on attribue à chaque lettre une valeur : **A** vaut 0, **B** vaut 1,..., **Z** vaut 25,
   - on prend la valeur de la lettre de rang 0 dans le mot, on multiplie par 26^0,
   - auquel on ajoute la valeur de la lettre de rang 1, multipliée par 26^1,
   - ...
   - auquel on ajoute la valeur de la lettre de rang k, multipliée par 26^k,
   - ...
   - de plus les calculs se font modulo l'entier p donné, donc la somme totale est un entier h avec $0 \leqslant h < p$.

   Programme une fonction `hachage(mot,p)` qui renvoie le hachage du mot donné modulo p.

   Exemples. Voici le détail des calculs du hachage de **LAPIN** modulo $p = 10$:

lettre	**L**	**A**	**P**	**I**	**N**
valeur	11	0	15	8	13
facteur	26^0	26^1	26^2	26^3	26^4
produit	11	0	10 140	140 608	5 940 688
modulo $p = 10$	1	0	0	8	8

   Donc le hachage de **LAPIN** modulo 10 est $h = 1 + 0 + 0 + 8 + 8 \pmod{10} = 17 \pmod{10} = 7$.

   Autre exemple avec **CHIEN** :

lettre	**C**	**H**	**I**	**E**	**N**
valeur	2	8	8	4	13
facteur	26^0	26^1	26^2	26^3	26^4
produit	2	182	5 408	70 304	5 940 688
modulo $p = 10$	2	2	8	4	8

   Donc le hachage de **CHIEN** modulo 10 est $h = 2 + 2 + 8 + 4 + 8 \pmod{10} = 24 \pmod{10} = 4$.

   Vérifie que le hachage de **SINGE** modulo 10 vaut aussi $h = 4$.

   Important. Il peut donc y avoir deux mots différents qui ont la même valeur de hachage.

   Indications. Pour récupérer le rang d'une lettre, définis une chaîne contenant toutes les lettres :

   ```
   ALPHABET = "ABCDEFGHIJKLMNOPQRSTUVWXYZ"
   ```

   Puis utilise la méthode `index()`, pour déterminer le rang d'un caractère stocké dans la variable `c` :

   ```
   ALPHABET.index(c)
   ```

   Par exemple `ALPHABET.index('D')` renvoie 3 (la numérotation commence à 0).

2. **Table de hachage et élimination de certains doublons.** Maintenant que l'on a associé un entier à chaque élément de la liste alors le principe est assez similaire à la première méthode, mais avec une table (dite table de hachage) beaucoup plus petite.

   Voici l'algorithme :
   - on fixe un entier p,
   - au départ on définit une table `table` qui contient p chaînes vides `''` (une pour chaque valeur de hachage possible entre 0 et $p-1$),
   - pour chaque `mot` de la liste initiale on calcule son hachage h (qui dépend du mot et de p) et on regarde `table[h]`, la valeur de la table au rang h :
     — s'il y a une chaîne vide `''` c'est que le mot est nouveau, on le conserve dans une nouvelle liste et en plus on place ce mot au rang h de la table : `table[h] = mot` ;

— s'il y a déjà une chaîne non-vide :
  — soit c'est la même chaîne que `mot`, auquel cas `mot` est un doublon que l'on ne conserve pas,
  — soit c'est une chaîne différente, auquel cas on conserve le mot.

Programme cet algorithme en une fonction `enlever_des_doublons(liste,p)` qui renvoie la liste originale des mots sans les doublons détectés par la fonction de hachage modulo p.

Prenons l'exemple de la liste `['CHIEN', 'LAPIN', 'CHIEN', 'SINGE', 'SINGE']` avec les calculs modulo $p = 10$.

- Au départ la table de hachage contient 10 chaînes vides :
  `['', '', '', '', '', '', '', '', '', '']`
- La hachage de **CHIEN** modulo 10 vaut $h = 4$ (voir la question précédente), donc on place ce mot dans la table au rang 4, la table est maintenant :
  table : `['', '', '', '', 'CHIEN', '', '', '', '', '']`
  et la nouvelle liste : `['CHIEN']`
- La hachage de **LAPIN** modulo 10 vaut $h = 7$, donc on place ce mot dans la table au rang 7, la table est maintenant :
  table : `['', '', '', '', 'CHIEN', '', '', 'LAPIN', '', '']`
  nouvelle liste :`['CHIEN', 'LAPIN']`
- Le mot suivant est encore **CHIEN**, de hachage $h = 7$. Il y a déjà un mot au rang 7, et comme c'est déjà **CHIEN** alors on ne change pas la table et on ne retient pas ce mot (rien ne change) :
  table : `['', '', '', '', 'CHIEN', '', '', 'LAPIN', '', '']`
  nouvelle liste :`['CHIEN', 'LAPIN']`
- Le mot suivant est **SINGE**, de hachage $h = 4$. Il y a déjà un mot au rang 4, mais c'est le mot **CHIEN**, comme ces deux mots sont différents alors on ne change pas la table mais on conserve le mot **SINGE** :
  table : `['', '', '', '', 'CHIEN', '', '', 'LAPIN', '', '']`
  nouvelle liste :`['CHIEN', 'LAPIN', 'SINGE']`
- Le mot suivant est encore **SINGE**, de hachage $h = 4$. Pour la même raison on conserve encore le mot **SINGE** :
  table : `['', '', '', '', 'CHIEN', '', '', 'LAPIN', '', '']`
  nouvelle liste :`['CHIEN', 'LAPIN', 'SINGE', 'SINGE']`

Notre méthode a donc supprimé un doublon de **CHIEN**, mais pas le doublon de **SINGE**. L'explication est simple : deux mots identiques ont bien la même valeur de hachage, mais il se peut que cela se produise aussi pour deux mots différents ! En choisissant p assez grand, ces accidents deviennent assez rares.

Par exemple ici **CHIEN** et **SINGE** ont la même valeur de hachage $h = 4$ pour $p = 10$, mais si on fixe $p = 11$ alors le hachage de **CHIEN** vaut $h = 2$ et celui de **SINGE** vaut $h = 5$.

3. **Itération et élimination de tous les doublons.**

   Programme une fonction `iterer_enlever_des_doublons(liste,nb_iter)` qui itère la fonction précédente avec différentes valeurs de p :
   - commence avec p qui vaut deux fois la longueur de la liste,
   - applique la fonction précédente avec cette valeur de p,
   - à partir de la liste renvoyée, retire les doublons avec cette fois $p \leftarrow p + 1$,
   - itère en incrémentant p à chaque fois.

   En général, avec trois itérations il y a peu de chance qu'il reste des doublons !

Exemple. Reprenons la liste :

`['CHIEN','LAPIN','CHIEN','SINGE','SINGE']`

Il y a 5 mots, donc on fixe $p = 10$. On retire d'abord les doublons modulo $p = 10$, on obtient la liste :

`['CHIEN','LAPIN','SINGE','SINGE']`

On repart de cette liste et on retire maintenant les doublons modulo $p = 11$, on obtient une liste sans doublons :

`['CHIEN','LAPIN','SINGE']`

Encore une fois, il serait facile d'effectuer le travail en parallèle sur les éléments : calcul du hachage et test s'il est présent dans la table.

Activité 3 (Calculs en parallèle sur les listes)**.**

Objectifs : voir des algorithmes bien adaptés aux calculs en parallèle pour les listes. Les fonctions de cette activités sont des fonctions récursives.

1. **Maximum d'une liste.**

   Il est facile de trouver le maximum d'une liste est parcourant la liste du début à la fin. On suppose ici que l'on a deux processeurs (ou plus). Voici l'idée pour profiter des calculs en parallèle : on divise la liste en deux, on cherche le maximum de la partie gauche (avec des processeurs), on cherche le maximum de la partie droite (avec d'autres processeurs) et on compare les deux maximums pour renvoyer le résultat.

   Voici l'algorithme qui définit une fonction récursive `maximum()` renvoyant le maximum d'une liste de nombre.

   Algorithme.
   - — Entête : `maximum(liste)`
     — Entrée : `liste`, une liste de n nombres.
     — Sortie : le maximum de la liste.
     — Action : fonction récursive.
   - **Cas terminaux.**
     — Si $n = 0$, renvoyer une très grande valeur négative (par exemple -1000).
     — Si $n = 1$, renvoyer le seul élément de la liste.
   - **Cas général où $n \geqslant 2$.**
     — Diviser la liste en deux parties : une partie gauche `liste_gauche` et une partie droite `liste_droite`.
     — Calculer le maximum de la partie gauche par un appel récursif `maximum(liste_gauche)`.
     — Calculer le maximum de la partie droite par un appel récursif `maximum(liste_droite)`.
     — Renvoyer le plus grand de ces deux maximums (à l'aide de la fonction habituelle `max(a,b)`).

   L'infini. La valeur -1000 du cas terminal est une valeur qui doit être plus petite que toute les valeurs de la liste. Il n'est donc pas sûr que -1000 soit suffisant. La bonne solution est d'utiliser la valeur infinie `inf` (qui correspond à $+\infty$) disponible depuis le module `math`. Par exemple « `3 < inf` » vaut « Vrai ». De même pour $-\infty$, « `3 > -inf` » vaut « Vrai ».

2. **Termes pairs d'une liste.**

   Il s'agit de ne conserver que les termes pairs d'une liste d'entiers. Par exemple à partir de la liste $[7, 2, 8, 12, 5, 8]$, on ne conserve que $[2, 8, 12, 8]$. L'idée pour profiter des calculs en parallèle est

similaire à celle de la question précédente : on divise la liste en deux, on cherche les termes pairs de la partie gauche, on cherche les termes pairs de la partie droite et on concatène ces deux listes.

Écris en détails l'algorithme récursif correspondant à cette idée. Réfléchis-bien aux cas terminaux (si la liste est vide bien sûr, on renvoie la liste vide, si la liste ne contient qu'un élément, que renvoie-t-on ?).

Programme ton algorithme en une fonction `extraire_pairs(liste)`.

3. **Premier rang non nul.**

   On considère une liste qui contient beaucoup de 0. Il s'agit de trouver le rang du premier terme non nul. Par exemple pour la liste $[0,0,0,0,0,1,0,1,1,0]$ le rang du premier terme non nul est 5 (on commence à compter à partir du rang 0).

   Voici l'idée pour profiter des calculs en parallèle :

   - on sépare la liste en une partie gauche et une partie droite,
   - par un appel récursif sur la partie gauche, on sait s'il y a un élément non nul dans la partie gauche, si c'est le cas on renvoie le rang et c'est fini, si ce n'est pas le cas on passe à la suite avec la partie droite,
   - si l'étude de la partie gauche n'a rien donné, on effectue un appel récursif sur la partie droite, s'il y a un élément non nul dans la partie droite, on renvoie le rang augmenté de la longueur de la liste de gauche et c'est fini.

     Écris en détails l'algorithme et programme une fonction `premier_rang(liste)`.

     Indications.

     — La fonction renvoie `None` si tous les éléments sont nuls.

     — Cas terminaux : si la liste est vide, on renvoie `None` ; si la liste ne contient qu'un élément, on renvoie `None` si cet élément est nul, et le rang 0 sinon.

     — Si l'étude de la sous-liste de gauche n'a rien donné il ne faut pas oublier de décaler le rang du premier terme non nul de la sous-liste de droite.

     — Par exemple pour la liste $[0,0,0,0,0,1,0,1,1,0]$, la partie gauche $[0,0,0,0,0]$ (de longueur 5) a tous ses termes nuls, pour la partie droite $[1,0,1,1,0]$ le premier terme non nul est de rang 0, mais dans la liste de départ ce terme est de rang $5+0=5$.

Cours 3 (Sommes partielles)**.**
Soit une suite d'éléments :

$$\big[x_0,\ x_1,\ x_2,\ \ldots,\ x_{n-1}\big],$$

la liste des ***sommes partielles*** est :

$$\big[x_0,\ x_0+x_1,\ x_0+x_1+x_2,\ \ldots,\ \ x_0+x_1+x_2+\cdots+x_{n-1}\big]$$

Exemples. La liste $\big[1,2,3,4,5,6,7,8\big]$ a pour sommes partielles

$$\big[1,3,6,10,15,21,28,36\big].$$

Autre exemple, la liste $\big[10,4,0,2,1,0,3,21\big]$ a pour sommes partielles $\big[10,14,14,16,17,17,20,41\big]$.

La k-ème somme partielle est donc :

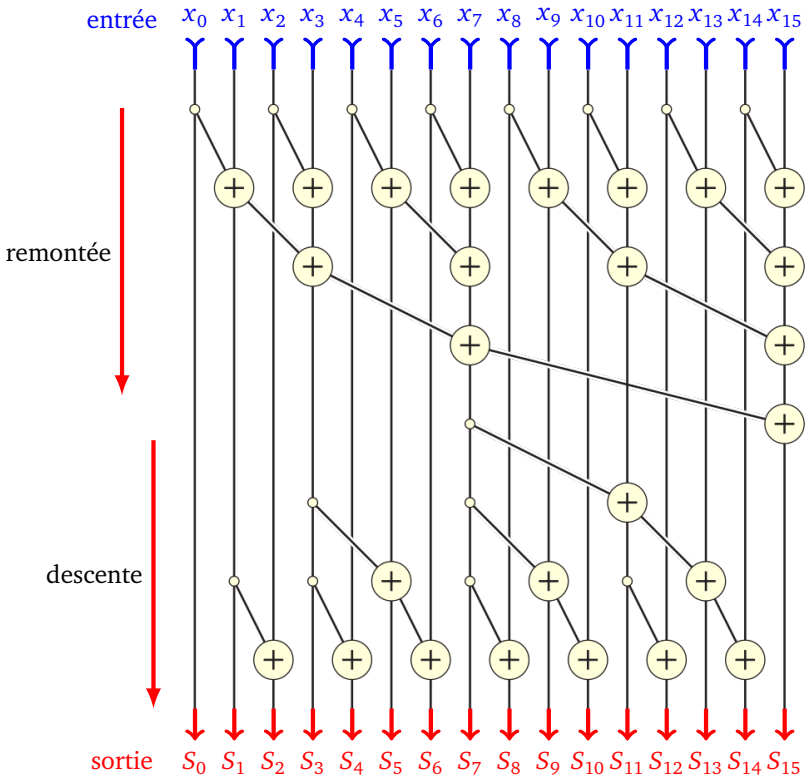
$$S_k = x_0+x_1+x_2+\cdots+x_k$$

et s'obtient à partir de la précédente par la formule

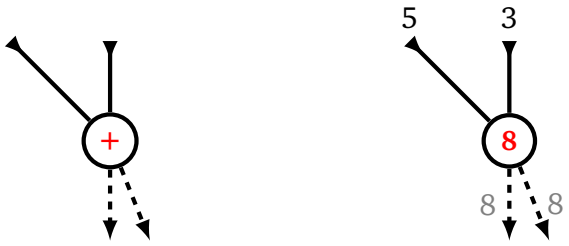
$$S_k = S_{k-1}+x_k$$

en initialisant $S_0 = x_0$ (ou mieux $S_{-1}=0$).
Calculs en parallèle.

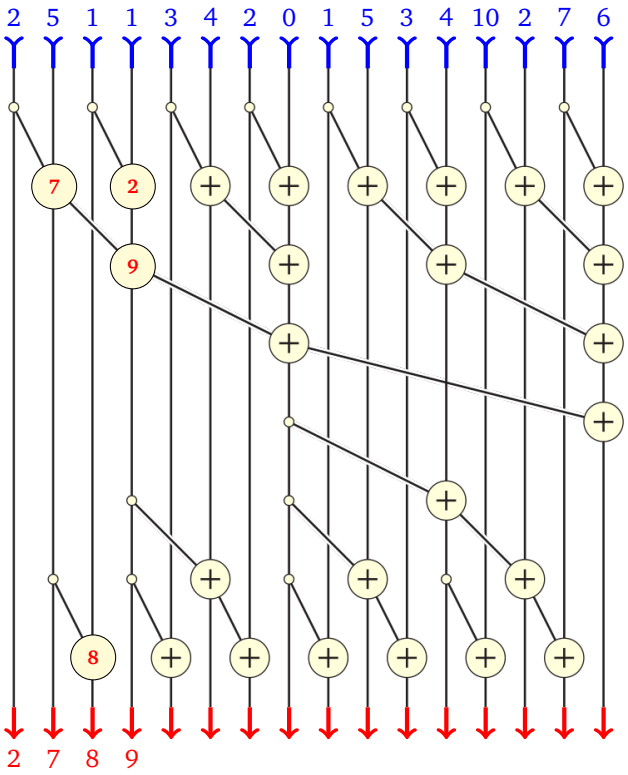
Il existe une méthode astucieuse pour effectuer les calculs en parallèle. Le principe est expliqué sur la figure ci-dessous.



Le schéma se lit de haut en bas. En haut il y a les entrées, ici une liste de 16 éléments. En bas la liste des sommes partielles. Chaque nœud ⊕ signifie l'addition des deux nombres issus des arêtes au-dessus. Tout ce qui sort vers le bas du nœud est le résultat de cette addition.



Exemple. Voici le début des calculs, à toi de les finir !



Activité 4 (Sommes partielles)**.**

Objectifs : il est très facile de calculer les sommes partielles par un algorithme séquentiel, cependant il existe un algorithme récursif qui permet de faire ces calculs en parallèle.

1. **Algorithme séquentiel.** Programme une fonction `sommes_partielles(liste)` qui renvoie la liste des sommes partielles après avoir parcouru (une seule fois !) la liste élément par élément.
2. **Algorithme parallèle.** Implémente l'algorithme suivant (qui correspond aux explications données ci-dessus) en une fonction récursive `sommes_partielles_recursif(liste)`. On suppose que la longueur de la liste est une puissance de 2.

Algorithme.

- — Entête : `sommes_partielles_recursif(liste)`
  — Entrée : `liste`$= [x_0, x_1, \ldots, x_{n-1}]$ une liste de n nombres (n est une puissance de 2).
  — Sortie : la liste de ses sommes partielles.
  — Action : fonction récursive.
- **Cas terminal.**
  Si $n = 1$, renvoyer la liste (qui ne contient qu'un élément).
- **Cas général où $n \geqslant 2$.**
  Remontée.
  — Former la liste `sous_liste` de taille $n//2$ constituée de l'addition d'un terme de rang pair et d'un terme de rang impair :
  $$[x_0 + x_1, x_2 + x_3, \ldots]$$
  — Avec cette `sous_liste`, faire un appel récursif :
  `sommes_partielles_recursif(sous_liste)`
  et nommer le résultat `liste_remontee` qui est une liste $[y_0, y_1, \ldots, y_{n//2-1}]$.
  Descente.
  — Initialiser une liste `liste_descente` qui contient seulement x_0.
  — Pour i allant de 1 à $n-1$:
  — Si i est pair, ajouter à `liste_descente` l'élément
  $$y_{i//2-1} + x_i,$$
  — sinon, ajouter à `liste_descente` l'élément
  $$y_{(i-1)//2}.$$
- Renvoyer la liste `liste_descente`.

3. **Applications : sélection des éléments par un filtre.**

Imaginons une liste quelconque, par exemple :

```
liste = [15, 18, 16, 11, 15, 19, 13, 12]
```

dont on souhaite ne conserver que certains éléments. Pour cela on définit un filtre :

```
filtre = [0, 0, 1, 0, 1, 0, 0, 1]
```

On ne retient que ceux qui sont en correspondance avec un 1 :

```
selec = [16, 15, 12]
```

C'est très facile à programmer de façon séquentielle (fais-le) et même en parallèle si on ne se préoccupe pas de l'ordre des éléments (on autorise la sortie `[15, 12, 16]` par exemple). Voici un algorithme pour le faire en parallèle en préservant l'ordre des éléments.

- On calcule la liste des sommes partielles du filtre. Sur notre exemple, cela donne :
  ```
  sommes = [0, 0, 1, 1, 2, 2, 2, 3]
  ```
  Le dernier élément n (ici $n = 3$) donne la taille de la liste finale `selec`. On initialise donc une liste `selec` de taille n.
- Les sommes partielles donnent le rang des éléments à conserver dans la liste finale (avec un décalage de 1). Comment ? Par exemple on sait qu'il faut conserver l'élément de rang 2 de la liste initiale, en effet pour $i = 2$ on a `filtre[i]=1` (et pas 0), donc il faut garder l'élément `liste[i]` qui vaut 16. La somme partielle au rang 2, `sommes[i]`, vaut 1, alors 16 aura comme rang final 0 (il y a un décalage de 1). Pour 15, sa somme partielle associée est 2, il sera donc au rang 1 ; pour 12, sa somme partielle est 3 il sera donc au rang 2.

- Ainsi pour i indexant les éléments de la liste initiale, si on doit conserver l'élément de rang i, alors il sera au rang `sommes[i]-1` dans la liste finale. Autrement dit :
  ```
  selec[sommes[i]-1] = liste[i]
  ```
- Les éléments à conserver sont maintenant dans la liste `selec`.

Programme cet algorithme en une fonction `selection(liste,filtre)`. (Si tu utilises la fonction de la question précédente, les longueurs des listes doivent être des puissances de 2.)