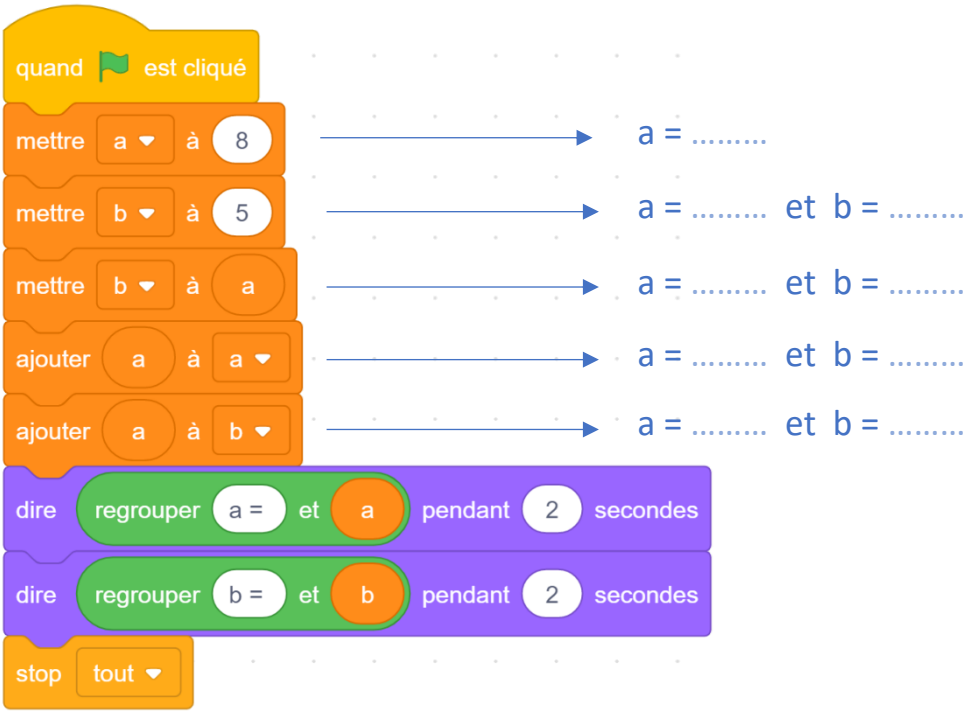
Le problème

Nous utilisons de nombreux programmes qui contiennent des variables. Il est donc essentiel de bien comprendre comment elles fonctionnent.

Analyse d'un programme

Saisir et tester le programme suivant. Pour chaque étape indiquer les valeurs des deux variables a et b une fois qu'elle a été exécutée.

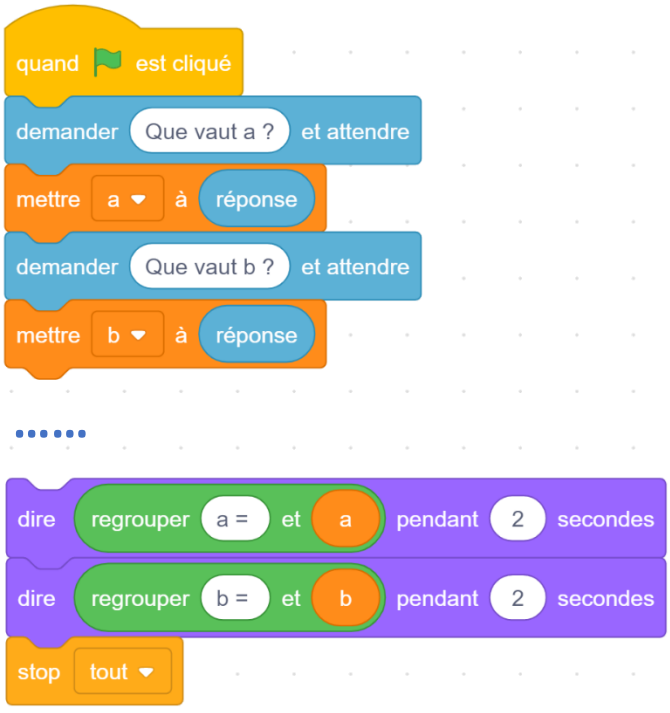


Programme à compléter

Compléter le programme ci-contre pour que les valeurs de a et de b affichées à la fin du programme soient une permutation de celle saisie au départ.

Par exemple :

Si l'utilisateur saisit $a = 5$ et $b = 8$ alors le programme aura permuté les deux valeurs et affichera $a = 8$ et $b = 5$.

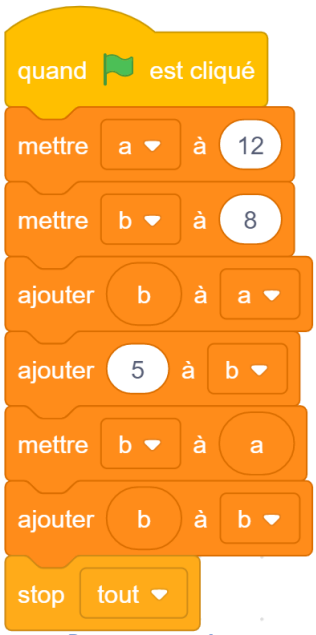


Le problème

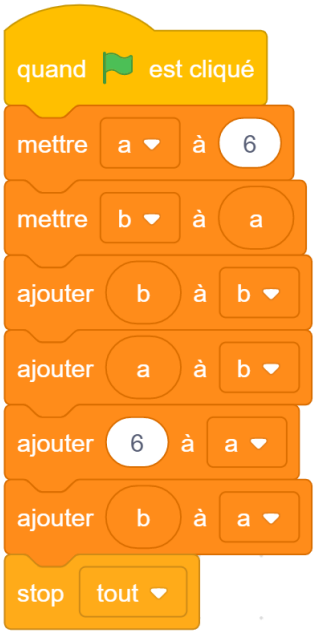
Nous utilisons de nombreux programmes qui contiennent des variables. Il est donc essentiel de bien comprendre comment elles fonctionnent.

Analyse de programmes

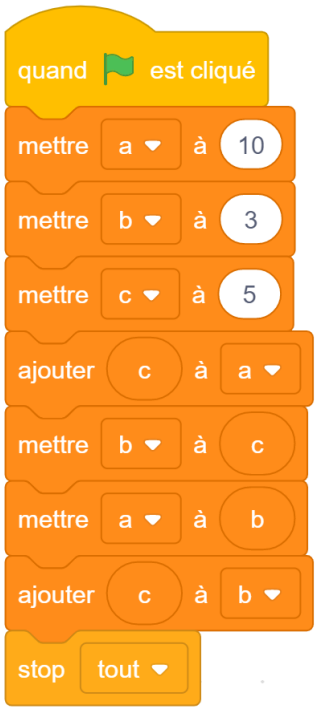
Pour chacun des programmes ci-dessous, indiquer les valeurs des variables à la fin du programme.



Programme 1

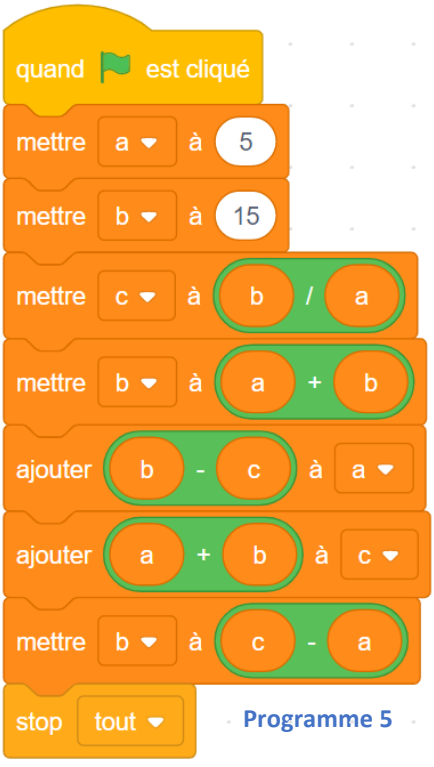


Programme 2

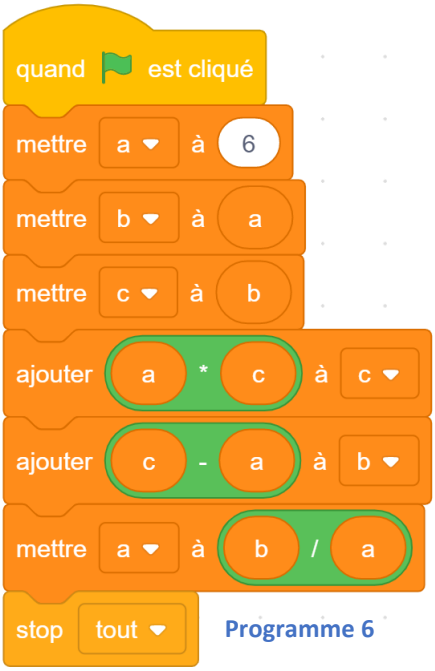


Programme 3

Programme 4



Programme 5



Programme 6

- À la fin du programme 1 :

  a = ... ; b = ...
- À la fin du programme 2 :

  a = ... ; b = ...
- À la fin du programme 3 :

  a = ... ; b = ... ; c = ...
- À la fin du programme 4 :

  a = ... ; b = ...; c = ...
- À la fin du programme 5 :

  a = ... ; b = ... ; c = ...
- À la fin du programme 6 :

  a = ... ; b = ... ; c = ...

Une somme d'inverses

Algorithmique

Le problème

Considérons la somme des inverses des premiers entiers naturels : $\frac{1}{1}+\frac{1}{2}+\frac{1}{3}+\frac{1}{4}+\frac{1}{5}+\frac{1}{6}+\frac{1}{7}+\ldots$

Nous voulons savoir s'il est possible que cette somme dépasse 10 et, si oui, combien de termes il faut ajouter pour y parvenir.

Avant de programmer

Calcule les sommes suivantes avec ta calculatrice (si nécessaire, donne une valeur approchée au centième) :

$\frac{1}{1}+\frac{1}{2}=$ .....

$\frac{1}{1}+\frac{1}{2}+\frac{1}{3}\approx$ .....

$\frac{1}{1}+\frac{1}{2}+\frac{1}{3}+\frac{1}{4}\approx$ .....

$\frac{1}{1}+\frac{1}{2}+\frac{1}{3}+\frac{1}{4}+\frac{1}{5}\approx$ .....

$\frac{1}{1}+\frac{1}{2}+\frac{1}{3}+\frac{1}{4}+\frac{1}{5}+\frac{1}{6}\approx$ .....

$\frac{1}{1}+\frac{1}{2}+\frac{1}{3}+\frac{1}{4}+\frac{1}{5}+\frac{1}{6}+\frac{1}{7}\approx$ .....

$\frac{1}{1}+\frac{1}{2}+\frac{1}{3}+\frac{1}{4}+\frac{1}{5}+\frac{1}{6}+\frac{1}{7}+\frac{1}{8}\approx$ .....

Cette somme augmente-t-elle rapidement ?

.....

D'après toi, combien de termes (environ) faudra-t-il pour qu'elle dépasse 10 ?

.....

Le programme à compléter

Ce programme utilise deux variables « dénominateur » et « somme » qui sont mises à 0 au début du programme.

La variable « dénominateur » prend la valeur du dénominateur du dernier terme de la somme. Il faudra donc l'incrémenter de 1 à chaque étape du calcul.

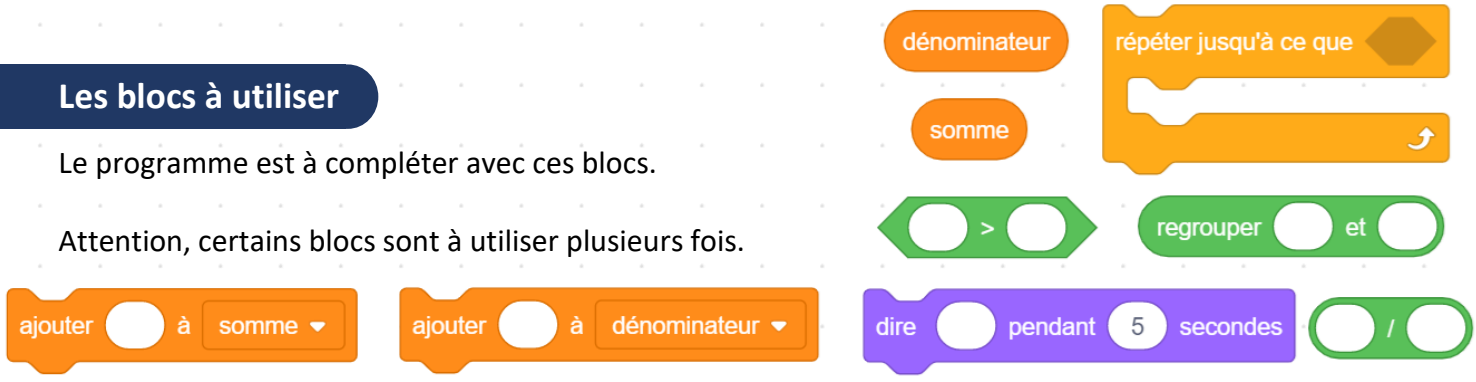
La variable « somme » prend la valeur de la somme calculée. Elle augmente donc à chaque étape du calcul.

Une fois le calcul terminé, le programme affiche le nombre de termes nécessaires.

Les blocs à utiliser

Le programme est à compléter avec ces blocs.

Attention, certains blocs sont à utiliser plusieurs fois.



Découverte de nouveaux blocs (2)

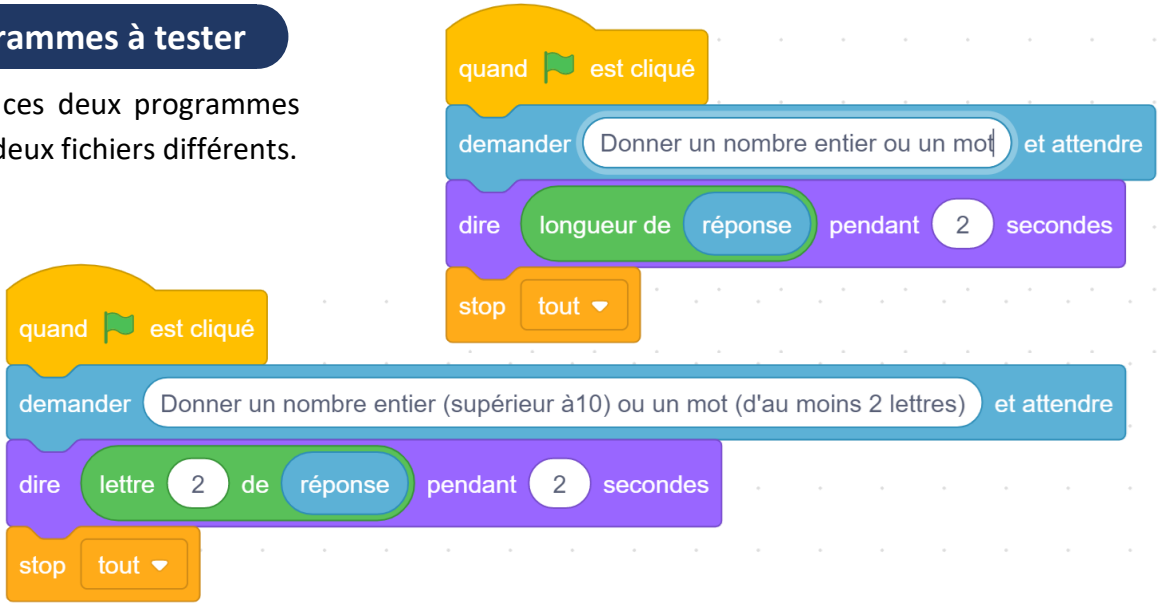
Algorithmique

Le problème

Certains blocs permettent d'obtenir des informations sur un nombre (ou mot) donné. Ils nous seront utiles pour réaliser des programmes plus complexes que ceux déjà réalisés.

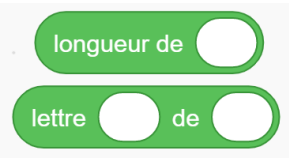
Programmes à tester

Saisir ces deux programmes dans deux fichiers différents.



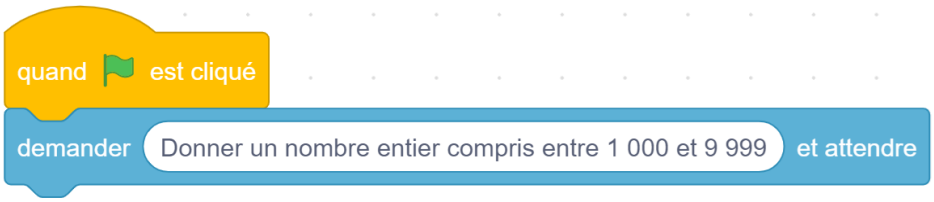
A quoi servent ces blocs ?

En testant ces programmes plusieurs fois avec différentes valeurs, expliquer la fonction des deux blocs ci-contre.



Programme à compléter

Complète le programme ci-contre pour qu'il donne le chiffre des unités du nombre choisi par l'utilisateur.



La réponse sera sous la forme d'une phrase complète. Par exemple, si le nombre choisi est 1 589, la réponse sera : « le chiffre des unités de 1 589 est 9 ».

Blocs à utiliser

Le programme est à compléter avec ces blocs. Attention, certains blocs sont à utiliser plusieurs fois.

Pour aller plus loin

Écris un programme qui donne le chiffre des unités de n'importe quel nombre entier donné.

Décomposition décimale d'un nombre entier à 4 chiffres

Algorithmique

Le problème

Nous voulons écrire un programme qui donne la décomposition décimale d'un nombre entier à quatre chiffres. Par exemple, si le nombre choisi est 9 527, le programme donnera « 9527 = 9 × 1000 + 5 × 100 + 2 × 10 + 7 ».

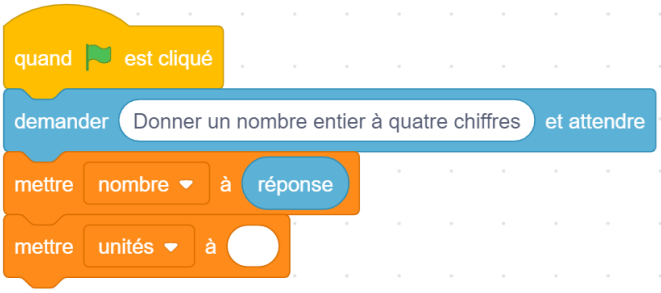
Avant de programmer

Donne la décomposition décimale de chacun des nombres suivants :

- 8 637 = ...
- 2 003 = ...
- 6 590 = ...

Programme à compléter

Voici les 4 premiers blocs du programme à compléter.



Ce programme utilise 5 variables :
« nombre », « unités », « dizaines », « centaines » et « unités de mille ».

La variable « nombre » a pour valeur le nombre choisi par l'utilisateur (il est préférable de ne pas travailler avec la variable « réponse »). Les quatre autres variables ont pour valeur les chiffres de ce nombre.

Blocs à utiliser

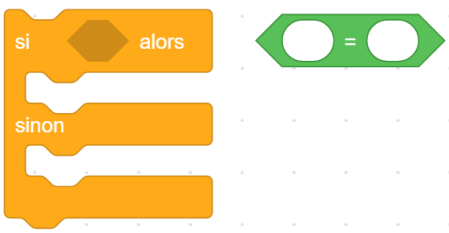
Le programme est à compléter avec ces blocs. Attention, certains blocs sont à utiliser plusieurs fois.

Pour aller plus loin

Améliore ce programme pour qu'il fonctionne, quel que soit le nombre entier choisi (inférieur à 9999). Avec le programme actuel, si le nombre choisi est 85 alors le résultat obtenu est « 85 = 0 × 1000 + 0 × 100 + 8 × 10 + 5 ».

Il s'agit de trouver un moyen pour ne pas afficher, dans ce cas, les termes « 0 × 1000 » et « 0 × 100 ».

Il faudra donc utiliser les blocs ci-contre (et peut-être d'autres).



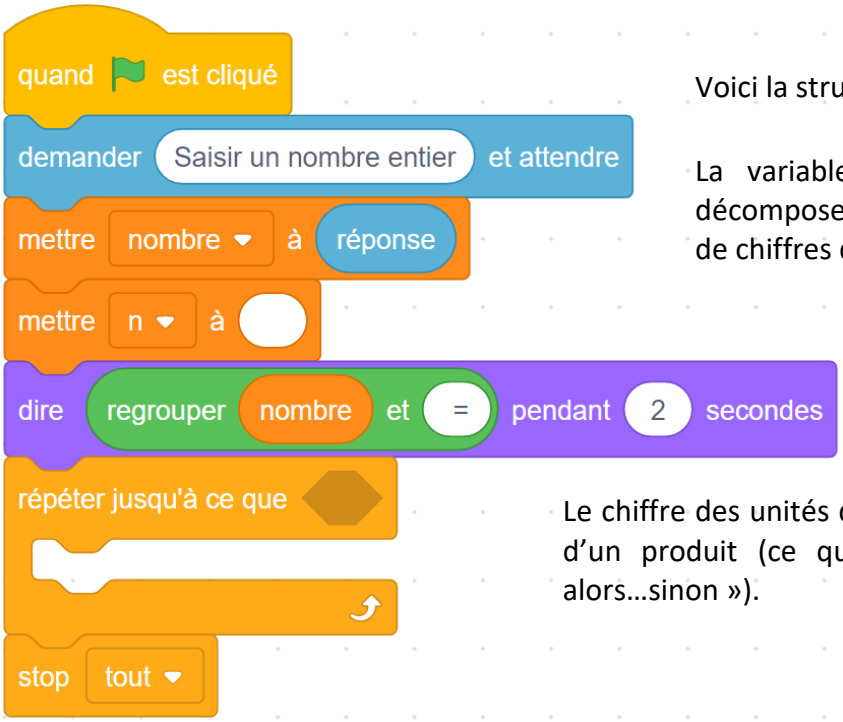
Décomposition décimale d'un nombre entier

Algorithmique

Le problème

Nous voulons écrire un programme qui donne la décomposition décimale d'un nombre entier donné, quel que soit son nombre de chiffres. Par exemple, si le nombre choisi est 67 638, le programme donnera « 67 638 = 6 × 10 000 + 7 × 1 000 + 6 × 100 + 3 × 10 + 8 ».
Les termes seront donnés successivement, car on ne sait pas à l'avance combien il y en aura.

Programme à compléter



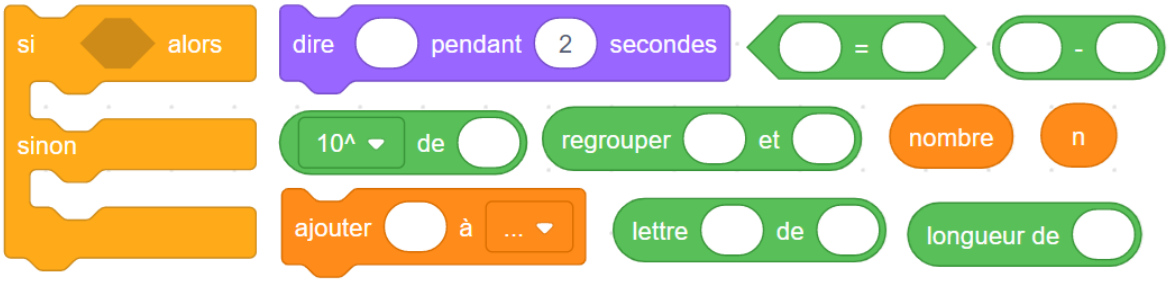
Voici la structure du programme à compléter.

La variable « nombre » contient le nombre à décomposer et la variable « n » contient le nombre de chiffres de ce nombre.

Le chiffre des unités doit être donné seul, et non sous forme d'un produit (ce qui justifie l'utilisation d'un bloc « si… alors…sinon »).

Blocs à utiliser

Le programme est à compléter avec ces blocs. Attention, certains blocs sont à utiliser plusieurs fois.



Pour aller plus loin

Améliore ce programme pour qu'il n'affiche pas les termes nuls. Par exemple, si le nombre saisi est 702, le programme ne devra pas afficher le terme « 0 × 10 ».

Une valeur approchée de π avec la formule de Wallis

Algorithmique

Le problème

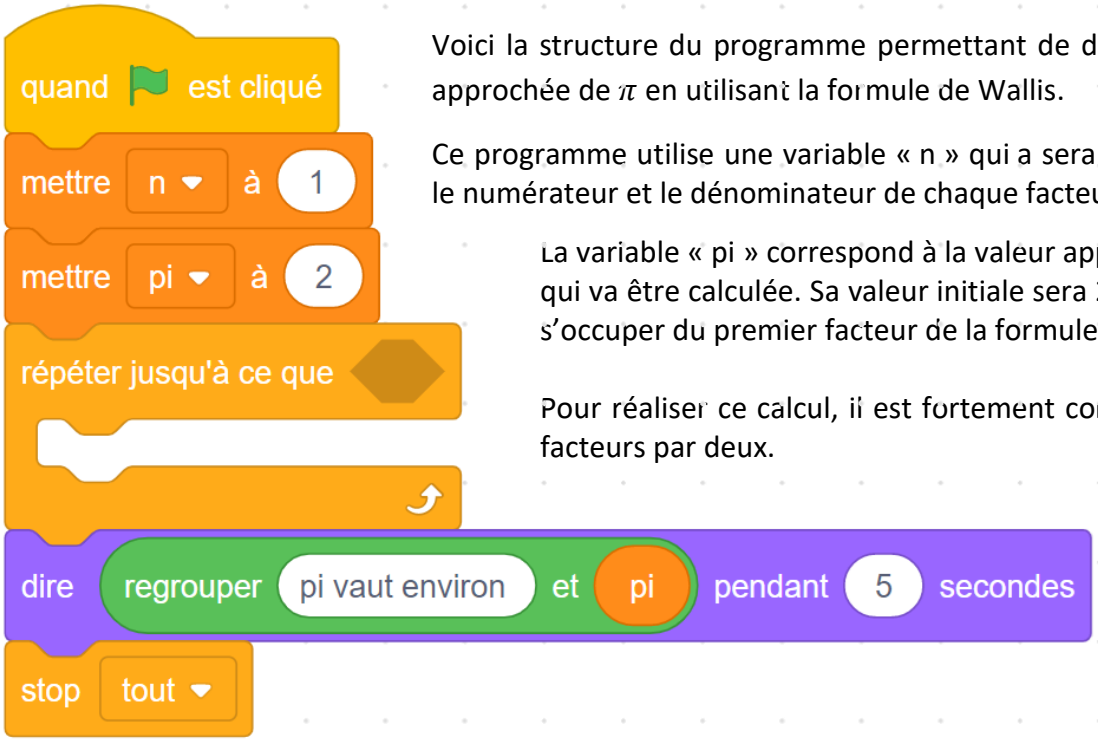
En 1655, le mathématicien John Wallis détermine par un calcul complexe une expression du nombre π sous forme d'un produit infini :

$$\pi = 2 \times \frac{2}{1} \times \frac{2}{3} \times \frac{4}{3} \times \frac{4}{5} \times \frac{6}{5} \times \frac{6}{7} \times \frac{8}{7} \times \frac{8}{9} \times \ldots$$

Ce calcul est toutefois peu utilisé, car il est peu performant : il faut utiliser un très grand nombre de facteurs pour déterminer seulement les premières décimales de π.

Nous voulons écrire un programme qui permette de calculer le produit des n premiers facteurs de ce produit (n étant un nombre entier déterminé) de façon à obtenir une valeur approchée du nombre π au millionième près ($\pi \approx 3{,}141592$ par défaut).

Structure du programme à compléter



Voici la structure du programme permettant de déterminer une valeur approchée de π en utilisant la formule de Wallis.

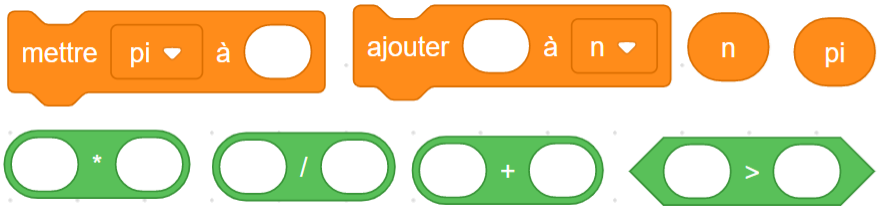
Ce programme utilise une variable « n » qui a sera utilisée pour calculer le numérateur et le dénominateur de chaque facteur du produit.

La variable « pi » correspond à la valeur approchée du nombre π qui va être calculée. Sa valeur initiale sera 2 : cela évite d'avoir à s'occuper du premier facteur de la formule.

Pour réaliser ce calcul, il est fortement conseillé de grouper les facteurs par deux.

Blocs à utiliser

Le programme est à compléter avec ces blocs. Attention, certains blocs sont à utiliser plusieurs fois.



Pour aller plus loin

Déterminer le nombre de facteurs nécessaire pour obtenir, avec cette formule, chacune des six premières décimales du nombre π.

Nombres palindromes (à quatre chiffres)

Algorithmique

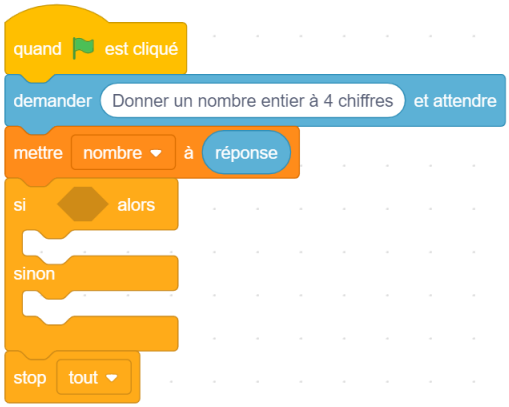
Le problème

On appelle nombre palindrome un nombre qui peut se lire de la gauche vers la droite ou de la droite vers la gauche.
Exemple : 494, 7447 et 16861 sont des nombres palindromes.
Nous souhaitons écrire un programme qui reconnaitra si un nombre à 4 chiffres donné est un nombre palindrome ou pas. Nous voulons ensuite écrire un autre programme qui déterminera le nombre de nombres palindromes à 4 chiffres.

Structure du premier programme à compléter

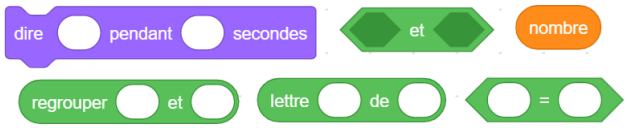
Voici la structure du programme permettant de dire si un nombre à 4 chiffres donné est un nombre palindrome ou pas.

Ce programme utilise une variable « nombre » qui a pour valeur le nombre choisi par l'utilisateur (il est préférable de ne pas travailler avec la variable « réponse »).



Blocs à utiliser

Le programme est à compléter avec ces blocs. Attention, certains blocs sont à utiliser plusieurs fois.



Structure du second programme à compléter

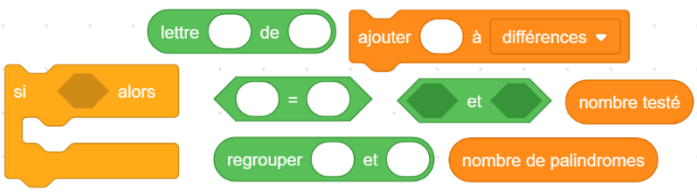
Voici la structure du programme permettant de déterminer le nombre de nombres palindromes à 4 chiffres.

Ce programme utilise une variable « nombre de palindromes » qui compte le nombre de nombres palindromes trouvé.

La variable « nombre testé » prend successivement toutes les valeurs de 1000 à 10000. Le programme doit vérifier, pour chacune de ses valeurs, si ce nombre est un nombre palindrome.

Blocs à utiliser

Le programme est à compléter avec ces blocs. Attention, certains sont à utiliser plusieurs fois.



Pour aller plus loin

Écris un programme permettant de déterminer le nombre de nombres palindromes à 5 chiffres.

Le problème

On appelle nombre palindrome un nombre qui peut se lire de la gauche vers la droite ou de la droite vers la gauche.

Exemple : 494, 7447 et 16861 sont des nombres palindromes.

Nous souhaitons écrire un programme qui déterminera si un nombre donné est un nombre palindrome ou pas.

Structure du programme à compléter

Voici la structure du programme permettant de déterminer si un nombre est un palindrome ou pas.

Ce programme utilise une variable « nombre » qui a pour valeur le nombre choisi par l'utilisateur (il est préférable de ne pas travailler avec la variable « réponse »).

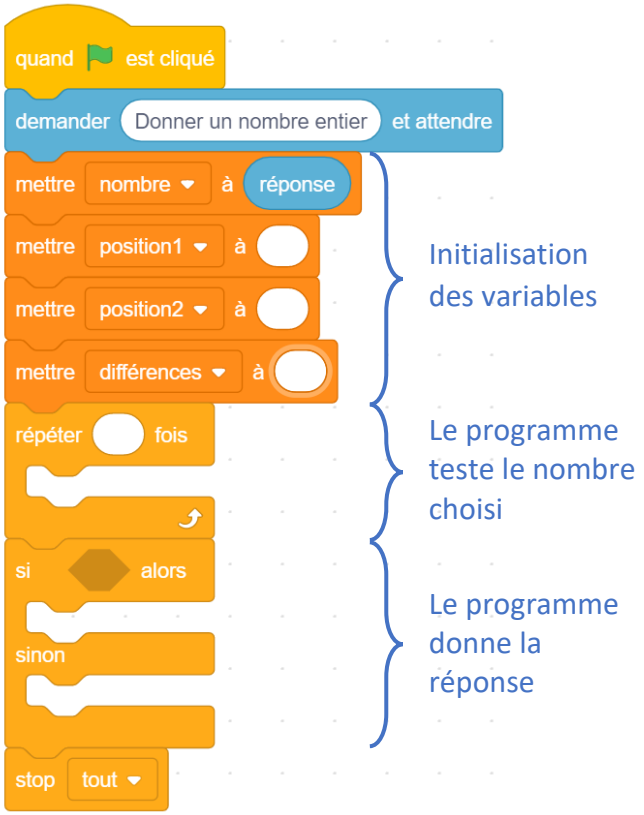
Les variables « position1 » et « position2 » donnent les rangs des chiffres à comparer :

4 8 6 1 6 8 4

Chiffre de rang « position 1 »

Chiffre de rang « position 2 »

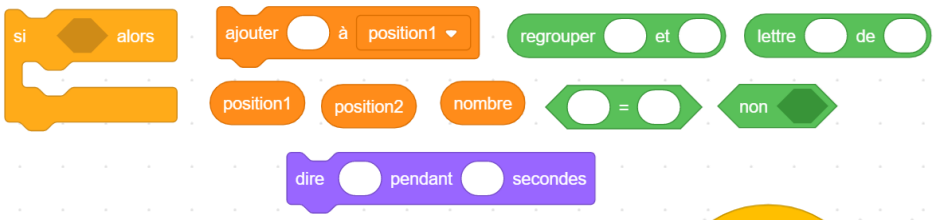
Après chaque test, « position1 » doit augmenter de 1 et « position2 » doit diminuer de 1.



La variable « différences » compte les différences entre les chiffres de rang « position1 » et ceux de rang « position2 ». Le nombre est un nombre palindrome s'il n'y a pas de différence.

Blocs à utiliser

Le programme est à compléter avec ces blocs. Attention, certains blocs sont à utiliser plusieurs fois.



Pour aller plus loin

Créer quatre blocs « question », « initialisation », « teste » et « résultats » pour que le programme principal soit semblable à celui ci-contre.

L'utilisation de tels blocs permet de mettre en évidence la décomposition d'un problème complexe en une succession de problèmes plus simples.

Le problème

Les listes sont un moyen puissant de stocker un grand nombre de données de façon ordonnée. Beaucoup de programmes informatiques les utilisent.

Programme à tester

Saisir et tester ce programme. Il faut créer la liste « Table de 5 ».
Expliquer le rôle de chacun des blocs fléchés :

Programme à créer

Commencer par créer un programme similaire au précédent qui donne dans une liste les 10 premiers nombres de la table de 6.

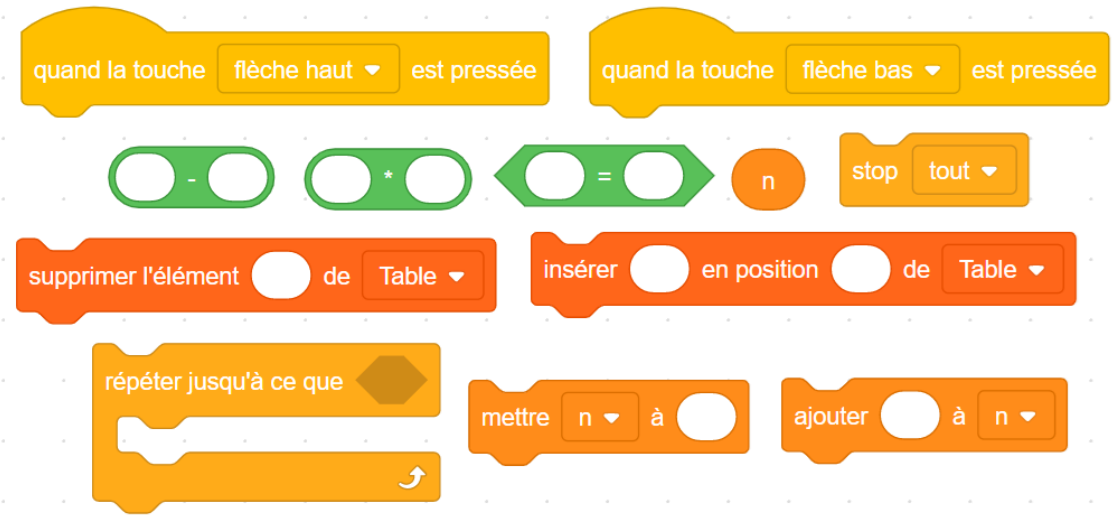
Ensuite, créer 2 sous-programmes tels que lorsque la liste de 6 est affichée, 2 actions sont possibles :

- Lorsque l'on appuie sur la flèche haut, on obtient la table de 12 en supprimant tous les nombres qui sont inutiles (on ne refait pas une liste, on supprime des éléments de la table de 6).
- Lorsque l'on appuie sur la flèche bas, on obtient la table de 3 en ajoutant tous les nombres qui manquent (on ne refait pas une liste, on complète la table de 6).

Conseil : il est plus simple de supprimer ou d'ajouter des éléments en commençant par la fin de la liste.

Blocs à utiliser

Les deux sous-programmes sont à créer avec ces blocs. Attention, certains blocs sont à utiliser plusieurs fois.



Pour aller plus loin

Faire en sorte que l'on puisse revenir à la table de 6 à partir des tables de 3 et de 12.

Ranger 3 nombres dans l'ordre croissant (1)

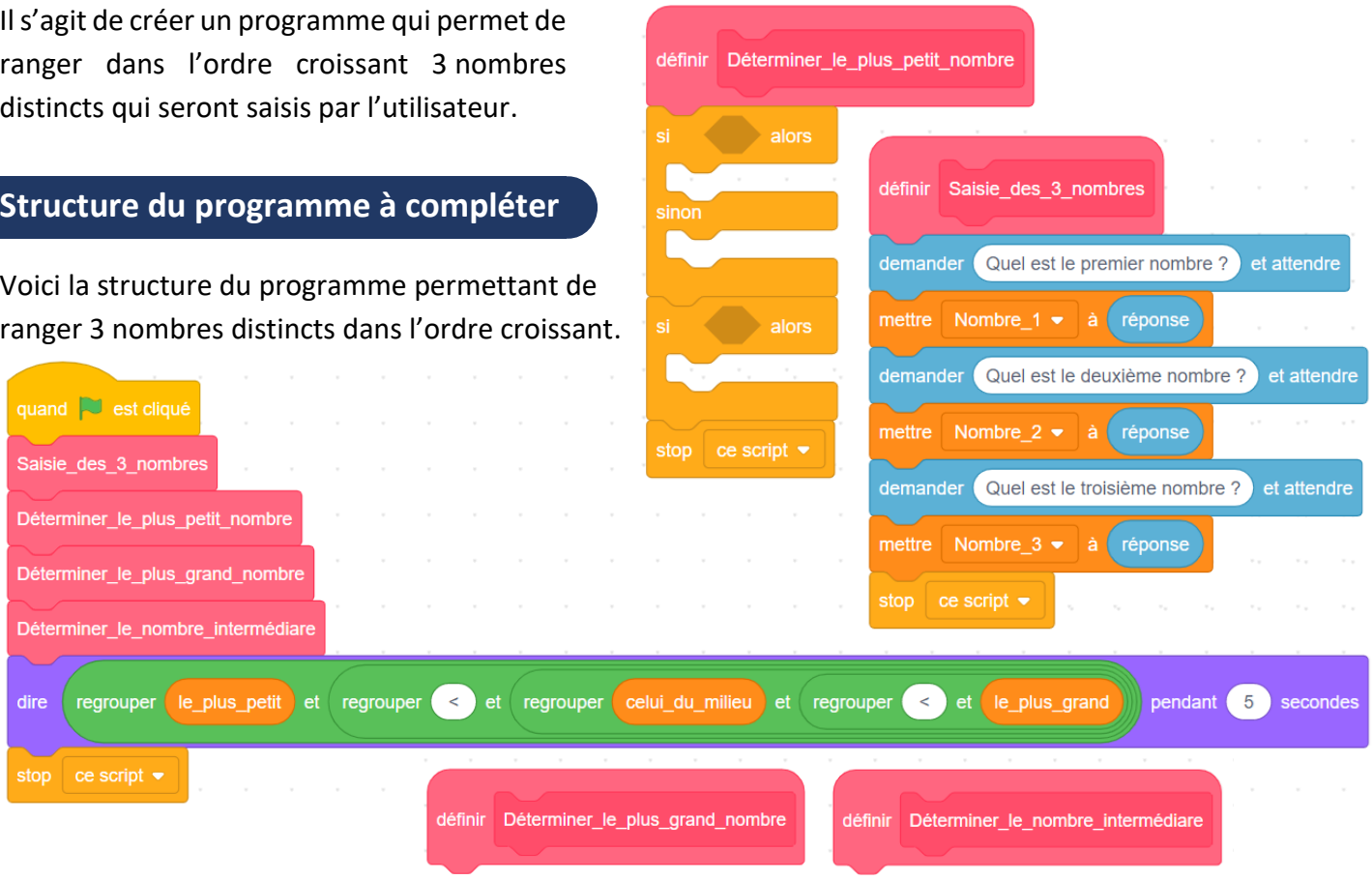
Algorithmique

Le problème

Il s'agit de créer un programme qui permet de ranger dans l'ordre croissant 3 nombres distincts qui seront saisis par l'utilisateur.

Structure du programme à compléter

Voici la structure du programme permettant de ranger 3 nombres distincts dans l'ordre croissant.



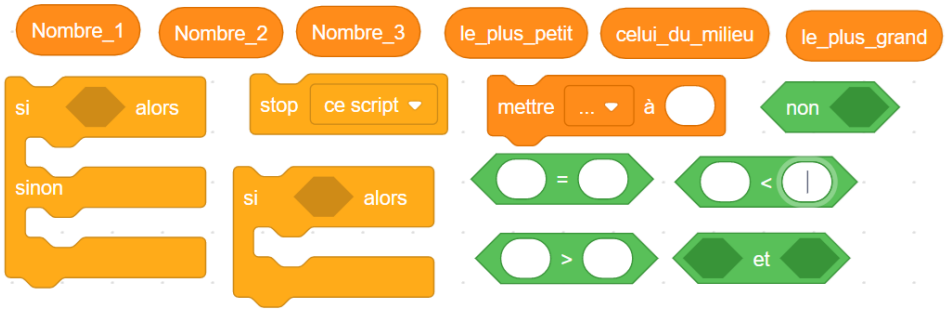
Ce programme utilise 6 variables : 3 pour les nombres saisis et 3 autres pour ces mêmes nombres rangés. **Les 3 variables contenant les 3 nombres à ranger ne sont pas modifiées dans ce programme : elles sont conservées**.

Le programme est décomposé en 4 fonctions :

- la saisie des 3 nombres à ranger (donnée) ;
- la détermination du plus petit de ces nombres (la structure est donnée) ;
- la détermination du plus grand de ces nombres (à définir entièrement) ;
- la détermination du nombre intermédiaire (à définir entièrement).

Blocs à utiliser

Le programme est à compléter avec ces blocs. Attention, certains blocs sont à utiliser plusieurs fois.



Pour aller plus loin

Dans la fonction de saisie, ajouter un message d'erreur si l'utilisateur choisit des nombres non distincts.

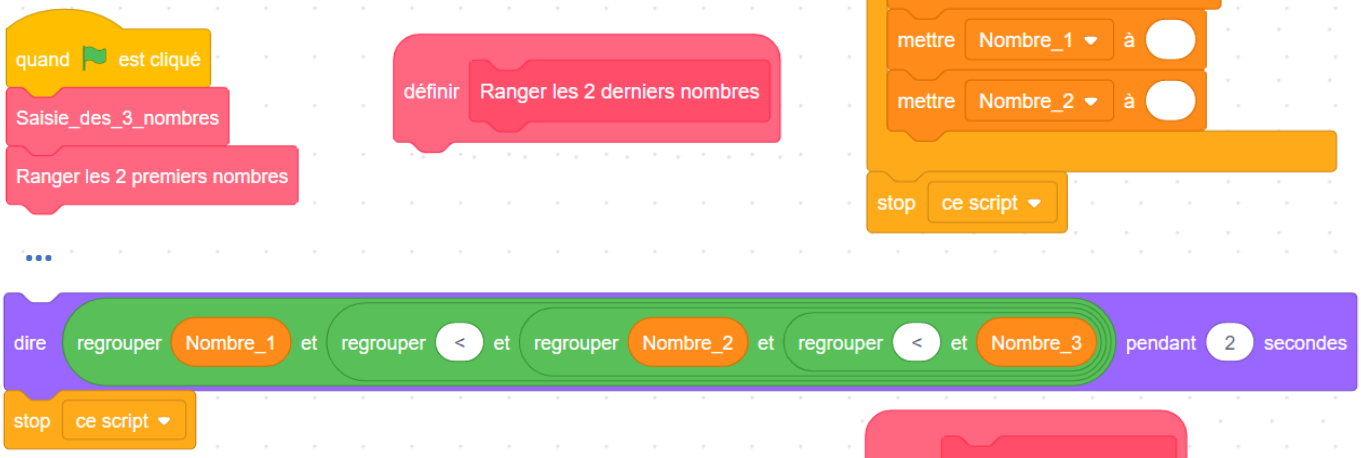
Ranger 3 nombres dans l'ordre croissant (2)

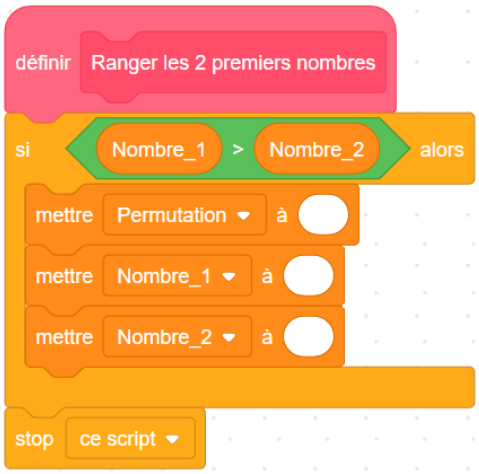
Le problème

Il s'agit de créer un programme qui permet de ranger dans l'ordre croissant 3 nombres distincts qui seront saisis par l'utilisateur.

Structure du programme à compléter

Voici la structure du programme permettant de ranger 3 nombres distincts dans l'ordre croissant.

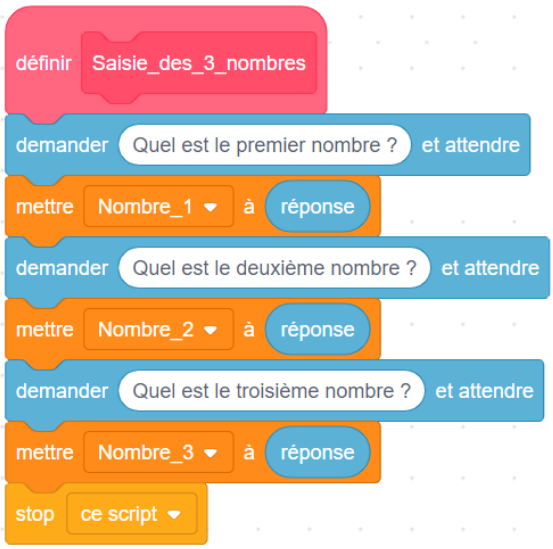




Ce programme utilise 4 variables : 3 pour les nombres saisis. **Ces 3 variables seront modifiées et rangées dans l'ordre à la fin du programme**. La 4e variable est utilisée pour réaliser les permutations.

Le programme est décomposé en 3 fonctions :

- la saisie des 3 nombres à ranger (donnée) ;
- une fonction pour ranger les 2 premiers nombres ;
- une fonction pour ranger les 2 derniers nombres ;



Blocs à utiliser

Le programme est à compléter avec ces blocs. Attention, certains blocs sont à utiliser plusieurs fois.

Pour aller plus loin

Compléter le programme pour pouvoir ranger 4 nombres dans l'ordre croissant.

Ranger 3 nombres dans l'ordre croissant (3)

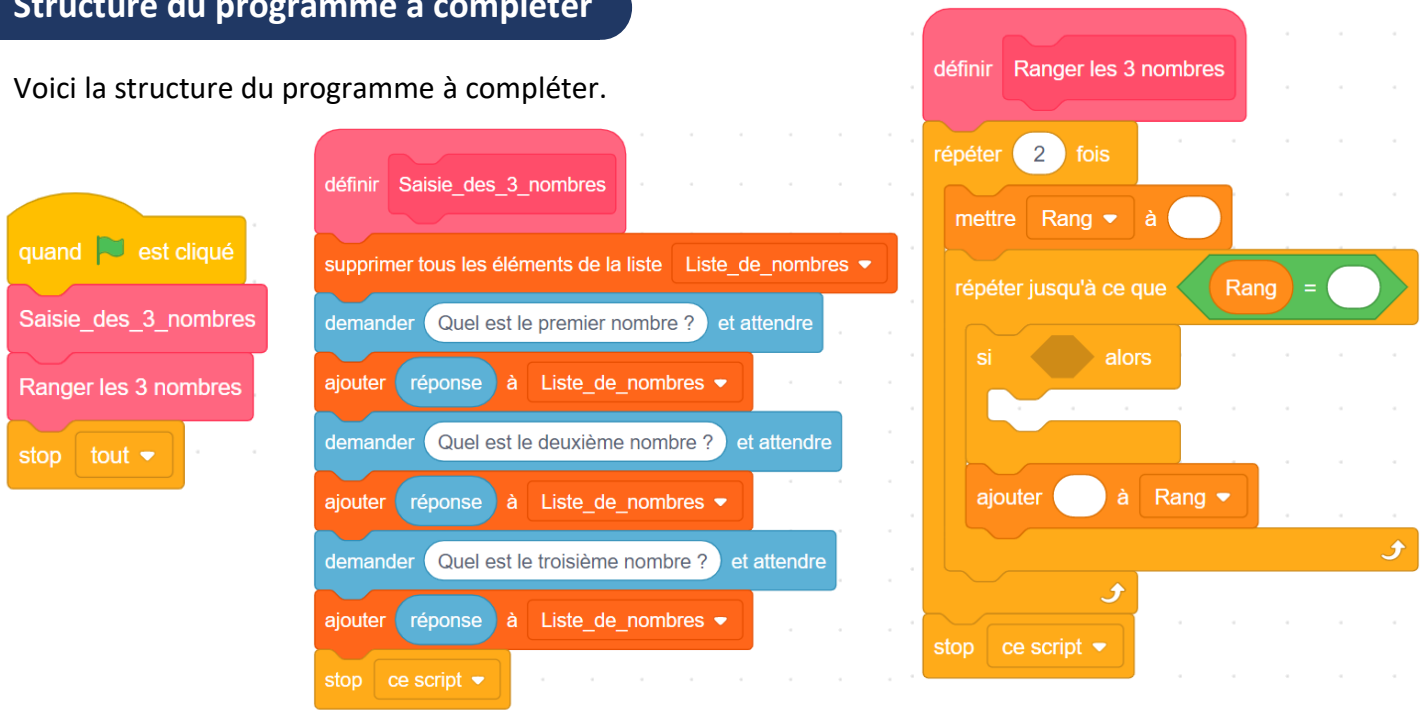
Algorithmique

Le problème

Il s'agit de créer un programme qui permet de ranger dans l'ordre croissant 3 nombres distincts qui seront saisis par l'utilisateur. Ce programme utilise une liste.

Structure du programme à compléter

Voici la structure du programme à compléter.



Les 3 nombres à ranger sont placés dans une liste.
La fonction « Ranger les 3 nombres » (à compléter) va comparer le 1er et le 2e nombre pour les ranger. S'il faut les permuter alors le 1er nombre est ajouté à la liste en 3e position puis le premier nombre de la liste est effacé.
Il fait ensuite de même avec le 2e et le 3e. Ce balayage doit être effectué 2 fois.

La variable « rang » sert de compteur. Elle contient la position dans la liste du nombre à éventuellement déplacer.

Blocs à utiliser

Le programme est à compléter avec ces blocs. Attention, certains blocs sont à utiliser plusieurs fois.

Pour aller plus loin

Compléter le programme pour pouvoir ranger 4 nombres dans l'ordre croissant. Pour ranger *n* nombres.

Déterminer une valeur approchée de $\sqrt{2}$

Algorithmique

Le problème

Il s'agit de créer un programme qui calcule une valeur approchée par défaut de $\sqrt{2}$ avec une précision donnée.

Structure du programme à compléter

Voici la structure du programme à compléter.

Le programme utilise 3 variables :

- La variable « Précision » est la précision du résultat attendu. Elle peut être modifiée avant de lancer le calcul.
- La variable « valeur » se rapproche à chaque étape un peu plus de la valeur recherchée.
- La variable résultat donnera **une valeur approchée par défaut** (à la précision donnée) de $\sqrt{2}$.

Le principe de ce programme est d'augmenter la variable « valeur » de la précision demandée jusqu'à obtenir une valeur plus grande que $\sqrt{2}$.

Blocs à utiliser

Le programme est à compléter avec ces blocs. Attention, certains blocs sont à utiliser plusieurs fois.

Pour aller plus loin

- Demander à l'utilisateur de saisir la précision.
- Demander à l'utilisateur de saisir le nombre dont on cherche la racine carrée.
- Combien de fois la boucle répéter est exécutée pour trouver le résultat ? Essayer de trouver un programme qui permet de limiter le nombre d'exécution de la boucle.

Déterminer une valeur approchée d'une racine carrée

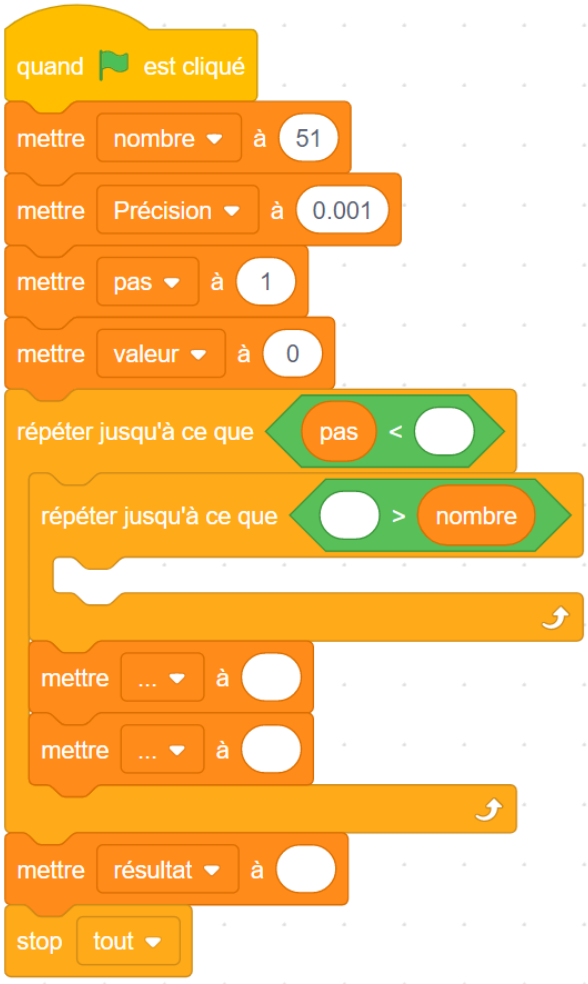
Algorithmique

Le problème

Il s'agit de créer un programme qui calcule une valeur approchée par défaut de la racine carrée d'un nombre avec une précision donnée mais en limitant le nombre de calculs effectués.

Structure du programme à compléter

Voici la structure du programme à compléter.



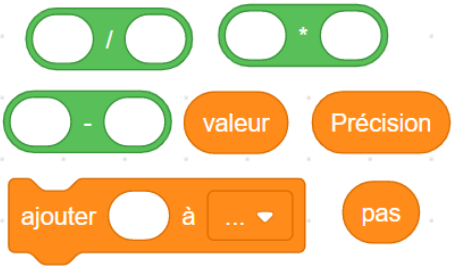
Le programme utilise 4 variables :

- La variable « nombre » contient le nombre dont on cherche la racine carrée (ici on cherche donc $\sqrt{51}$). Elle peut être modifiée avant de lancer le calcul.
- La variable « Précision » est la précision du résultat attendu. Elle peut être modifiée avant de lancer le calcul.
- La variable « valeur » se rapproche à chaque étape un peu plus de la valeur recherchée.
- La variable « pas » est la valeur qu'il faut ajouter à la variable « valeur » à chaque étape. On commence avec un pas de 1, puis on continue avec un pas de 0,1 puis de 0,01 : le pas est divisé par 10 tant qu'il est supérieur à la précision demandée).
- La variable résultat donnera **une valeur approchée par défaut** (à la précision donnée) de la racine carrée cherchée.

Le principe de ce programme est de tester les valeurs de 1 à 8 avec un pas de 1 (dans le cas de $\sqrt{51}$), puis de 7 à 7,2 avec un pas de 0,1, puis de 7,10 à 7,15 avec un pas 0,01…

Blocs à utiliser

Le programme est à compléter avec ces blocs. Attention, certains blocs sont à utiliser plusieurs fois.



Pour aller plus loin

- Demander à l'utilisateur de saisir le nombre dont on cherche la racine carrée.
- Demander à l'utilisateur de saisir la précision.

Le problème

Nous souhaitons écrire un programme qui déterminera si un nombre donné est un nombre premier ou pas. Pour cela le programme essayera de diviser ce nombre par tous les entiers de 1 jusqu'à lui-même.

Structure du programme à compléter

Voici la structure du programme permettant de dire si un nombre donné est un nombre premier ou pas.

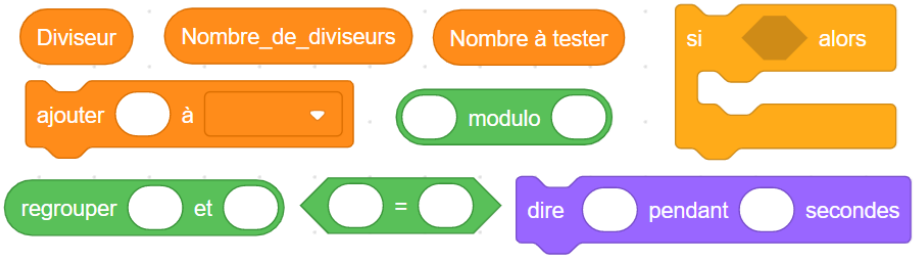
La variable « Nombre à tester » prend la valeur du nombre dont on veut savoir s'il est premier.

La variable « diviseur » va prendre successivement toutes les valeurs de 1 au nombre à tester. À chaque étape, le programme teste si la variable « Nombre à tester » est divisible par cette variable « diviseur ».

La variable « Nombre de diviseurs » compte le nombre de diviseurs trouvés. C'est ce nombre qui permettra de conclure si le nombre est premier ou pas.

Blocs à utiliser

Le programme est à compléter avec ces blocs. Attention, certains blocs sont à utiliser plusieurs fois.



Pour aller plus loin

Tester le programme avec le nombre 10 518 311 et chronométrer le temps mis pour obtenir la réponse.

Modifier le programme pour qu'il divise le nombre choisi par tous les entiers de 1 jusqu'à sa racine carrée. Tester le programme avec des nombres premiers connus pour vérifier qu'il fonctionne.

Chronométrer à nouveau le programme avec le nombre 10 518 311. Quel constat peut-on faire sur la nécessité d'optimiser un programme informatique ?

Liste des diviseurs d'un nombre

Algorithmique

Le problème

Nous souhaitons écrire un programme qui donnera la liste complète des diviseurs d'un nombre donné.

Structure du programme à compléter

Voici la structure du programme à compléter.

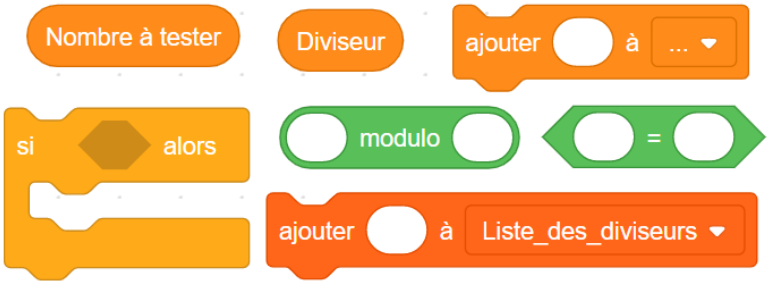
La variable « Nombre à tester » prend la valeur du nombre dont on veut connaitre les diviseurs.

La variable « diviseur » va prendre successivement toutes les valeurs de 1 au nombre à tester.

À chaque étape, le programme teste si la variable « Nombre à tester » est divisible par cette variable « diviseur ». A chaque fois que c'est le cas, le diviseur est ajouté à la liste « Liste des diviseurs »

Blocs à utiliser

Le programme est à compléter avec ces blocs. Attention, certains blocs sont à utiliser plusieurs fois.



Pour aller plus loin

- Faire dire au programme le nombre de diviseurs obtenus.
- Si le nombre testé est premier, le faire dire par le programme.
- Ajouter un message d'erreur si le nombre saisi n'est pas un nombre entier.

Multiplier sans multiplication

Algorithmique

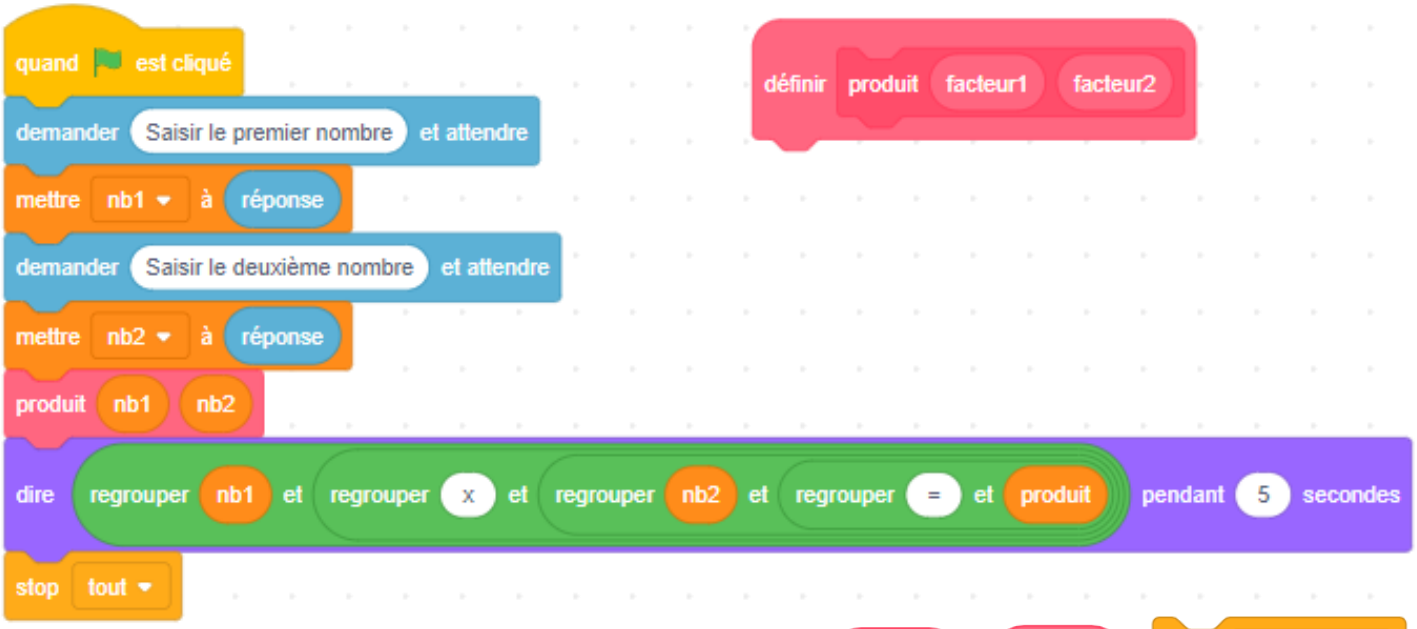
Le problème

Nous souhaitons écrire un programme qui calcule le produit de deux nombres entiers strictement positifs en utilisant uniquement des additions. Par exemple, 6 × 3 peut se calculer par 6 + 6 + 6 ou par 3 + 3 + 3 + 3 + 3 + 3.

Structure du programme à compléter

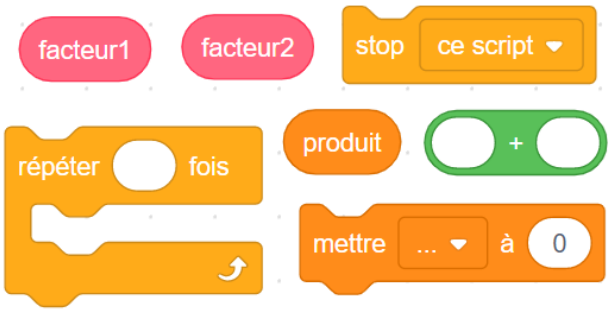
Voici la structure du programme à compléter.

Il faut définir la fonction « produit » qui contient deux arguments (les deux facteurs du produit à calculer). Le résultat du calcul sera contenu dans la variable « produit ».



Blocs à utiliser

Le programme est à compléter avec ces blocs. Attention, certains blocs sont à utiliser plusieurs fois.



Pour aller plus loin

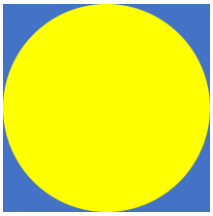
- Teste le programme avec 2 et 10 000 000 (dans cet ordre) puis avec 10 000 000 et 2 (dans cet ordre). Que constates-tu ?
- Ajouter au programme une fonction « ranger » pour savoir lequel est le plus petit des deux facteurs : utiliser deux variables « inf » et « sup ». Le calcul du produit se fera ensuite avec ces deux nouvelles variables.
- Compléter le programme pour qu'il fonctionne également avec des nombres entiers négatifs ou nuls.

Approximation de π avec la méthode de Monte-Carlo

Algorithmique

Le problème

La méthode de Monte-Carlo est une méthode de calcul qui utilise les probabilités. Elle a été inventée en 1947 par le physicien gréco-américain Nicholas Metropolis et permet, entre autres, de déterminer une approximation du nombre π.



Il s'agit de placer de façon aléatoire des points sur une cible formée d'un disque inscrit dans un carré. Si l'on place suffisamment de points (toute la surface de la cible est couverte) alors le quotient $\frac{nombre\ de\ points\ dans\ le\ disque}{nombre\ de\ points\ total}$ sera proche du quotient $\frac{Aire\ du\ disque}{Aire\ du\ carré}$.

Avant de programmer

Vérifier l'égalité : $\frac{Aire\ du\ disque}{Aire\ du\ carré} = \frac{\pi}{4}$ et en déduire une expression d'une valeur approchée de π en fonction du nombre de points dans le disque et du nombre total de points.

Structure du programme à compléter

Utiliser le fichier
https://scratch.mit.edu/projects/487730691
qui contient la cible.

Voici la structure du programme permettant de déterminer une approximation de π en utilisant la méthode de Monte-Carlo.

La variable « nombre de points » compte le nombre de points aléatoires qui sont placés et la variable « nombres de points dans le disque » compte, parmi ces points, combien sont situés dans le disque.

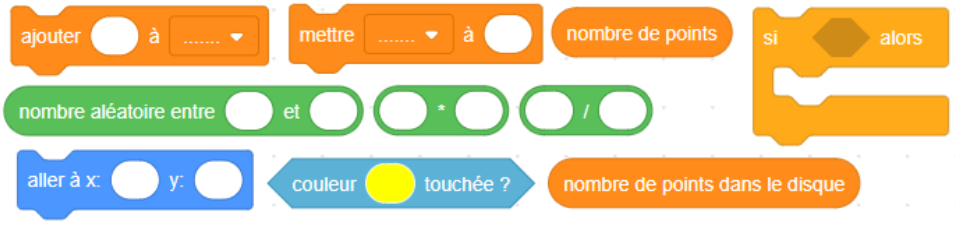
Une autre variable « PI » est l'approximation de π déterminée par la simulation.

Ces 3 variables sont actualisées à chaque nouveau point placé.

À chaque répétition, le lutin « Point » se voit attribuer des coordonnées aléatoires entre -180 et 180 (pour l'abscisse et l'ordonnée). Pour savoir si ce point est alors dans le disque, on effectue un test pour savoir s'il touche la couleur jaune du disque.

Blocs à utiliser

Le programme est à compléter avec ces blocs. Attention, certains blocs sont à utiliser plusieurs fois.



Pour aller plus loin

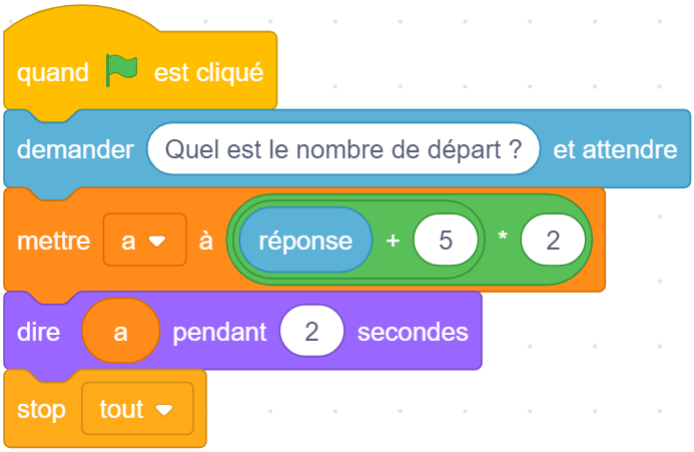
- Si on lance plusieurs fois le programme, obtient-on toujours le même résultat ? Expliquer pourquoi.
- Expliquer les raisons de l'imprécision des résultats obtenus avec ce programme.

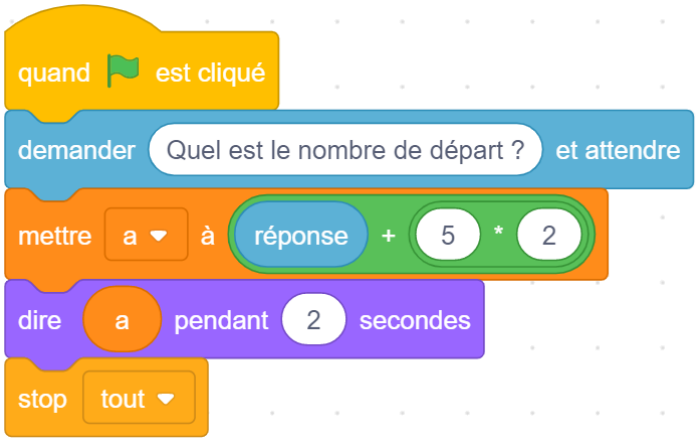
Le problème

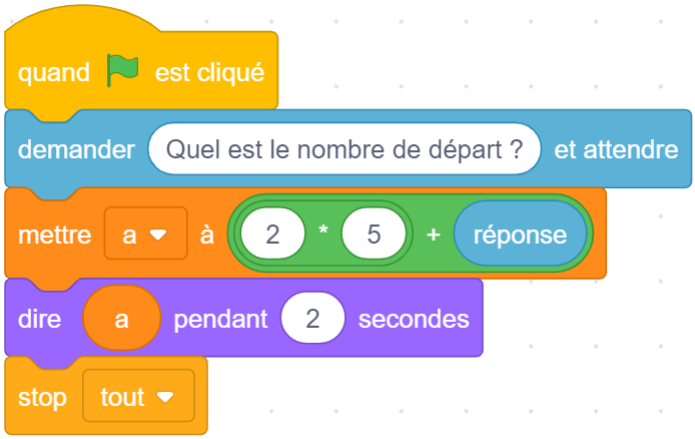
Dans certains programmes, nous aurons besoin de réaliser des séries de calculs. Pour cela, il est indispensable de comprendre comment Scratch gère les priorités opératoires.

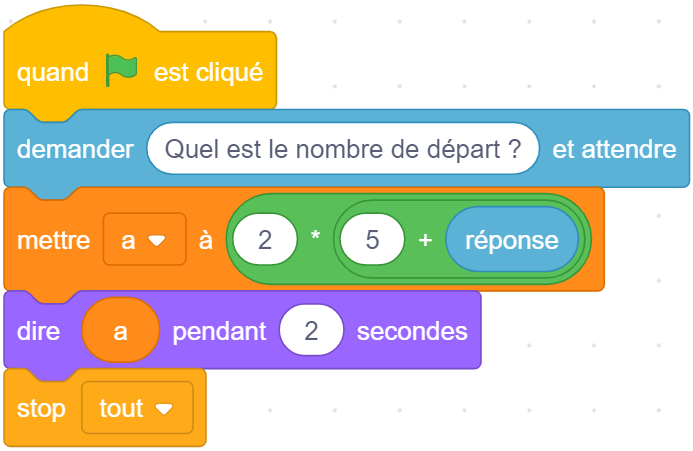
Programmes à tester

Voici les quatre programmes à tester. Ils utilisent une variable nommée « a ».







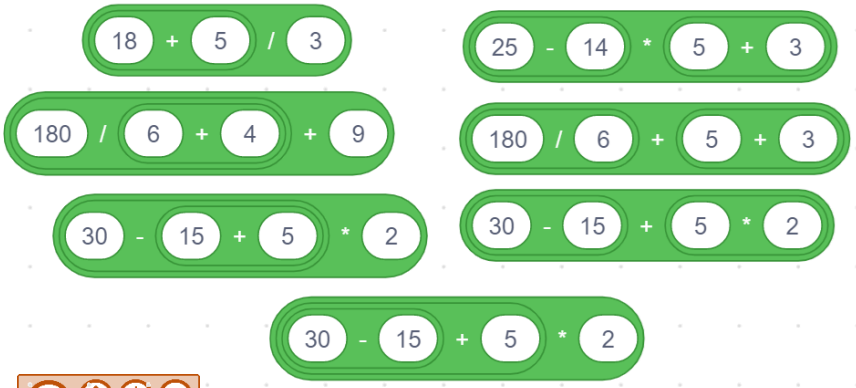


Priorités opératoires ?

En testant ces programmes plusieurs fois avec différentes valeurs, expliquer comment Scratch gère les priorités opératoires.

Applications

Que renvoie chacun de ces sept calculs ?
Tu peux les tester sur Scratch.



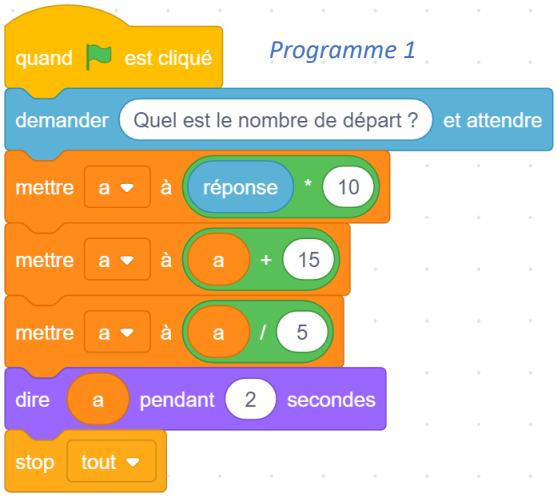
Le problème

Un programme de calcul est un algorithme présentant une succession de calculs à partir d'une valeur initiale. Le résultat obtenu dépend de cette valeur initiale. Il est important de comprendre un tel programme.

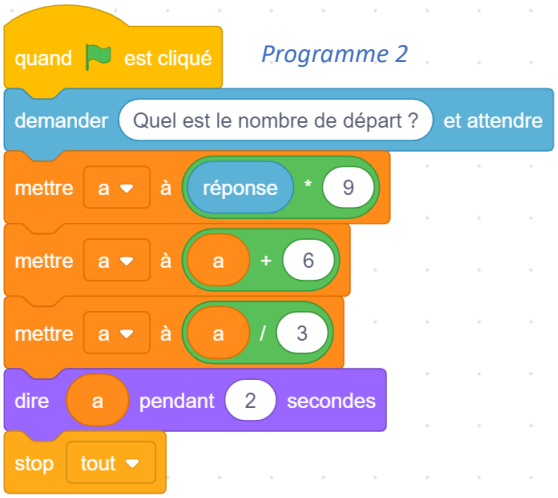
Programmes à interpréter

Pour chacun de ces programmes, donner le résultat affiché pour chacune de ces valeurs de départ : 0, 1, 2, 3, 5 et 11.

Programme 1

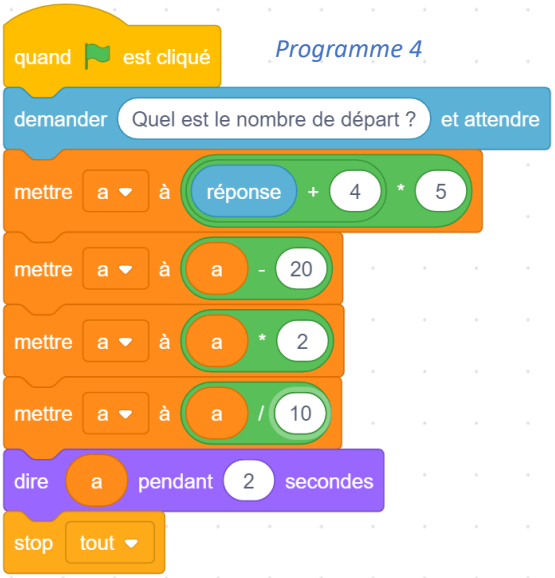


Programme 2



Programme 3

Programme 4



Simplification des programmes

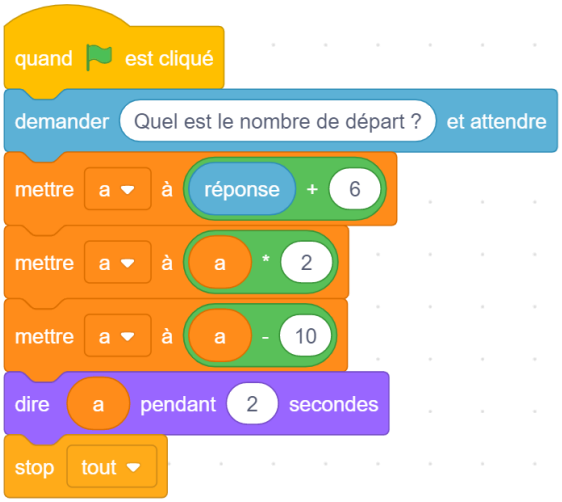
1. Que semble faire le programme 4 ? Essaye de le prouver avec une succession de calculs avec une variable a (ne pas lui donner de valeur).
2. Peut-on trouver un calcul plus simple pour les programmes 1 et 2 ? Si oui justifie le par une succession de calculs avec une variable a (ne pas lui donner de valeur).
3. Même question avec le programme 3.

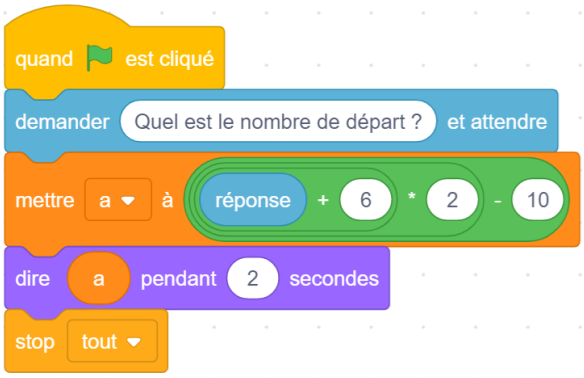
Le problème

Pour pouvoir simplifier certains programmes, il faut être capable d'écrire en une seule expression (une expression littérale) un programme de calcul écrit en plusieurs lignes (une ligne par étape).

Programmes à tester

Saisir et tester ces deux programmes pour différentes valeurs de départ (il faut créer la variable « a »). Que constates-tu ?





Réécriture des programmes

Suivant l'exemple ci-dessus, écris les deux programmes suivants en utilisant qu'une seule ligne de calcul (conserver tous les calculs).

Simplification des programmes

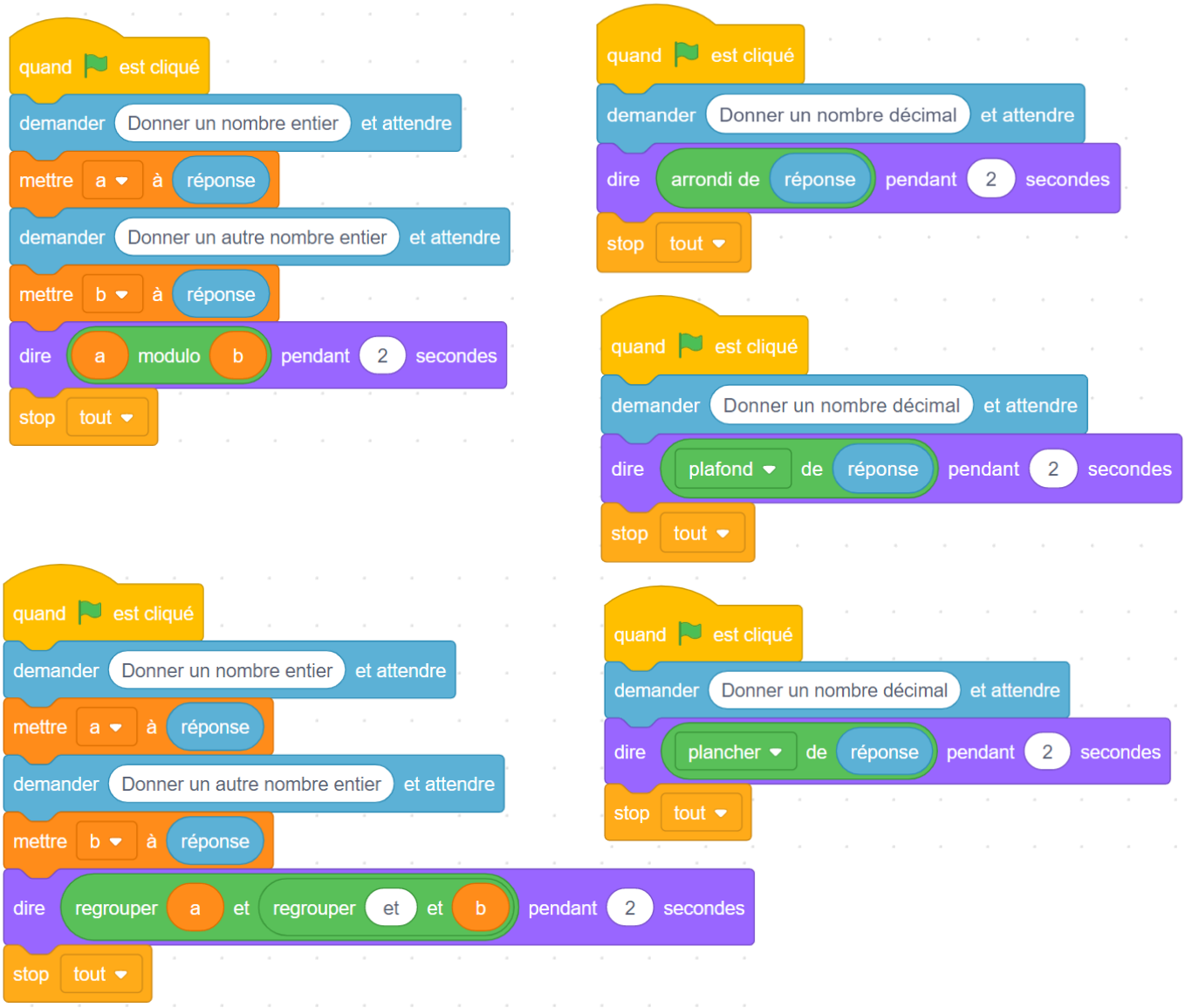
Que semblent faire les deux programmes précédents ? Essaye de le montrer avec une succession de calculs avec une variable a (ne pas lui donner de valeur).

Le problème

Pour réaliser des programmes plus complexes que ceux déjà réalisés, nous allons avoir besoin d'utiliser des blocs encore jamais étudiés et qu'il est important de connaitre.

Programmes à tester

Saisir ces cinq programmes dans cinq fichiers différents.



A quoi servent ces blocs ?

En testant ces programmes plusieurs fois avec différentes valeurs, expliquer ce que font les cinq blocs :

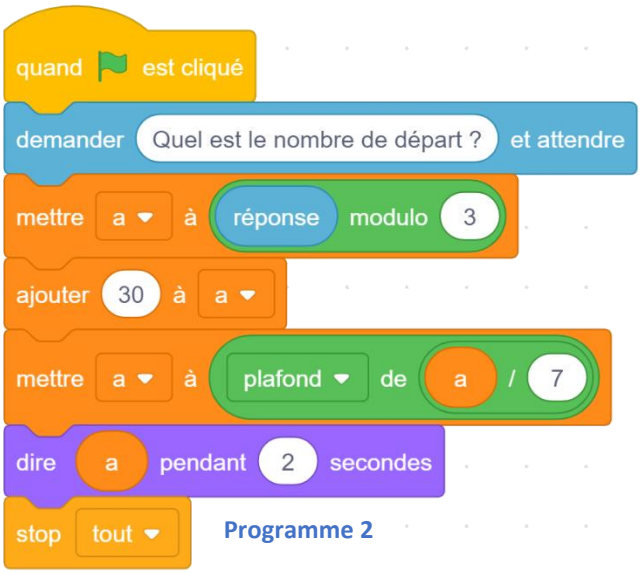
Le problème

Un programme de calcul peut contenir des opérateurs autres que les 4 opérations (+, -, ×, ÷). Il est important de connaitre tous les opérateurs pour pouvoir comprendre un tel programme.

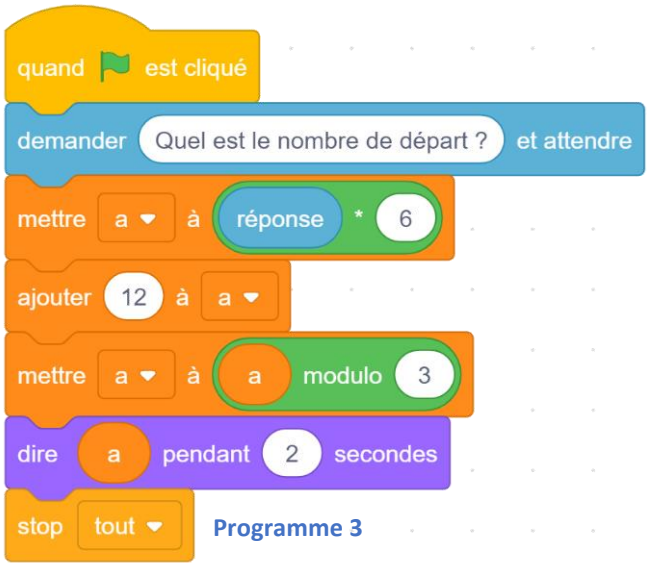
Analyse de programmes

Pour chacun de ces programmes, donner le résultat affiché pour chacune de ces valeurs de départ : 0, 1, 2, 5, 6 et 8.

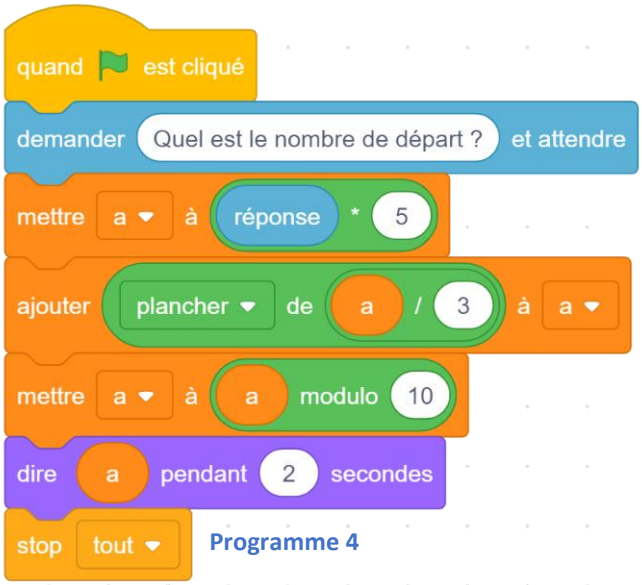
Programme 1



Programme 2



Programme 3



Programme 4

Simplification d'un programme

Que semble faire le programme 3 ? Essaye de le prouver.

Le problème

Nous voulons écrire un programme avec Scratch qui donnera l'égalité bilan de la division euclidienne d'un nombre a par un nombre b. Ces deux nombres étant saisis par l'utilisateur.

La structure du programme

Voici, ci-dessous, la structure du programme à réaliser.
Ce programme utilise 4 variables : « Dividende », « Diviseur », « quotient » et « reste ».

Le programme demande à l'utilisateur de saisir les valeurs du dividende et du diviseur.

Le programme calcule les valeurs du quotient et du reste.

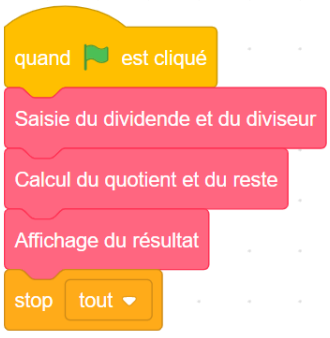
Le programme affiche l'égalité bilan de la division euclidienne demandée. Si le reste est nul, on ne l'affiche pas (c'est pour cela qu'il y a une condition).

Les blocs à utiliser

Le programme est à compléter avec ces blocs.

Attention, certains blocs sont à utiliser plusieurs fois.

Pour améliorer le programme



Créer trois nouveaux blocs pour que le programme se présente comme ci-contre.

Cette méthode permet de bien décomposer le problème de départ (qui est complexe) en sous-problèmes plus simples.