**Algorithme**

**Appel à une autre fonction**Spécifique au problème de la primalité  
Commentaires / Améliorations

**Main**

Si aucun fichier génération de départ n’existe, on en crée un en générant un fichier comportant 200 fonctions **mutation(vide)**.

Soit, on conserve toutes les générations dans des fichiers générationXXXX, soit, pour des raisons de places, on supprime les générations antérieures.

On boucle d’une génération à l’autre par **nextgen(n)** jusqu’à ce qu’on ait atteint le Score\_max = 256.  
Au lieu de s’arrêter une fois, le score\_max atteint, on préfèrera réduire ensuite en taille au maximum les fonctions.

**Nextgen**

On commence par **calculer** le score de chacune des fonctions de la génération parent.  
Si le score\_max est atteint, on s’arrête là.

On classe ensuite les fonctions, d’abord en fonction de leur score, puis, celles qui ont le score\_limite, en fonction de leur taille ou en privilégiant les nouveaux-nés (parmi les 100 dernières).

On copie, bien entendu, les 100 premières fonctions du classement ainsi établi.

Fonction après fonction dans le nouveau fichier, on va procéder avec un ratio de 1/10 à une **mutation** ou à un **crossover** (échange).

Ainsi la génération suivante, contenant 200 fonctions, est terminée.

**Calculer**

Dans le cadre d’une programmation en thread (processeurs à plusieurs cœurs), on va diviser cette fonction en, par exemple, 8 blocs, qui vont chacun calculer 1/8 du total.

On se contente de comparer les entiers de 2 à 255, écrits en nombre binaire à un tableau isprime, valant 1 si le nombre est premier et 0 sinon :  
On incrémente de 1 le score lorsque l’**évaluation** de la fonction en x vaut isprime[x].

**Evaluation**

On lit le string de la fonction de manière postfixée et on travaille par empilement/dépilement.  
Si on rencontre la lettre ‘x’ et que ‘x’ vaut x, on met x dans la pile.  
Si on rencontre la lette ‘X’ et que X à une arité de n, on retire a1,..,an, derniers éléments de la pile, et on y place X(a1,..,an), sachant comment calculer X(a1,..,an).  
Ainsi une fois le string totalement lu, il ne reste qu’un élément dans la pile : la valeur recherchée.

Partie du programme la plus coûteuse (lorsqu’écrit en C). Meilleure manière de procéder ?

C’est ici que sont définies les fonctions élémentaires E, O, N.

**Mutation**

On se place aléatoirement sur un caractère R du string de la fonction.  
Par calcul d’arité, on remonte le string jusqu’à avoir le sous-arbre ayant pour racine R.  
On supprime alors ce sous-arbre, et on y copie, à la place, une fonction élémentaire aléatoire ( qui peut être une variable ), ainsi que des variables aléatoires auxquelles elle s’applique.

**Crossover**

On récupère une fonction des 100 copiées dans la nouvelle génération et on lui prend un sous-arbre A.  
Comme pour la mutation, on va alors remplacer un sous-arbre de la fonction à modifier par A.

On a évidemment toutes les fonctions auxiliaires, qui permettent, entre autres, de manipuler les fichiers et d’en ressortir des strings, de binariser un entier, de générer la liste des nombes premiers jusqu’à 255 …

Une éventualité serait d’abandonner supprimer le temps de lecture/écriture en gardant deux générations en mémoire en permanence.  
Le problème est qu’il devient impossible de couper le programme en cours d’exécution sans risque de perte des données.  
Il s’est avéré, lorsque j’ai placé des timers pour voir ce qui prenait le plus de temps à l’exécution, que tout était négligeable devant le temps d’empilement/dépilement, je n’ai donc pas essayé de modifier ce procédé.