Autour des nombres premiers

Pour les questions de programmation, voir le fichier .py.

1 Préliminaires

1.
2.
3. Cet appel renvoie 3.
4. On constate que `mystere(x,b)` renvoie l'exposant maximal k tel que $b^k \leqslant n$.
   Cela revient à renvoyer l'entier k tel que $b^k \leqslant x < b^{k+1}$, c'est-à-dire $k \leqslant \log_b(x) < k+1$ c'est-à-dire $\log_b(x) - 1 < k \leqslant \log_b(x)$ c'est-à-dire $k = \lfloor \log_b x \rfloor$.
   Remarque : C'est le nombre de chiffres pour écrire n en base b moins 1.
   Remarque : Erreur d'énoncé : b doit être $\geqslant 2$ sans quoi la fonction peut ne pas terminer.
5. À la fin de la boucle `x1` contient $10^5 \times 10^{-5}$ et `x2` $\sum_{k=0}^{10^5-1} 10^{-5}$.
   Si les calculs étaient exacts les deux contiendrait 1. Mais ce n'est pas le cas car les nombres manipulés sont des flottants.

2 Génération de nombres premiers

2.1 Approche systématique

1. *Remarque :* Les tests que j'ai effectué semble montrer que Python utilise moins de 32 bits par cases pour enregistrer un tableau de booléens, mais bon, faisons comme si...

   Une mémoire vive de 4Go, soit $4 \times 10^9 \times 8$ bits permet d'enregistrer un tableau de $\dfrac{4 \times 10^9 \times 8}{32}$ cases, c'est-à-dire 10^9 cases.
2. Comme il n'y a que deux valeurs pour un booléen, on peut les coder par un seul bit. Ce qui permettrait un gain mémoire d'un facteur de presque 32 (presque car le coût du tableau lui-même reste le même).
3. *Remarque :* Il y a plusieurs difficultés techniques :
   - L'élément i est dans la case $i-1$ du tableau.
   - Boucle « pour » qui inclut les bornes : attention au passage au range.
   - Lorsque N est trop grand, `int(sqrt(N))` ne marche pas car passage en flottant. Je n'ai cependant pas traité ce problème dans le corrigé.
4. Soit p un nombre premier. La ligne « Marquer comme faux les multiples de p différents de p » s'exécute en $O\left(\frac{N}{p}\right)$ (si elle est bien codée...)

   La complexité totale est donc $\sum_{p \text{ premier} \leqslant N} O\left(\frac{N}{p}\right)$, qui vaut $_{N\to\infty} O\left(\sum_{p \text{ premier} \leqslant N} \frac{N}{p}\right)$ (série à termes positifs qui diverge), et donc $O_{N\to\infty}\big(N \log(\log(N))\big)$.
5. Soit b un base et n le nombre de chiffres de N. Alors $N = O(b^n)$. Si l'on veut faire des calculs avec les « grands O », il faudra utiliser des propriétés du genre « On peut passer au log dans des grand O si les suites sont strictement positives et tendent vers l'infini », propriétés pas spécialement citées dans les programmes officiels de maths et d'info. Je préfère utiliser de braves inégalités.
   Ainsi, je pars du fait que $N < b^n$ (car le plus grand entiers qu'on peut écrire avec n chiffres en base b a tous ses chiffres égaux à $b-1$, c'est donc $\sum_{i=0}^{n-1}(b-1)b^i$, soit $(b-1)\frac{1-b^n}{1-b} = b^n - 1$.)

   J'obtiens alors que $N \log(\log(N)) < b^n \log(n \log b) = O_{n\to\infty}(b^n \log n)$.

3 Génération rapide de nombres premiers

1. L'énoncé est particulièrement pas clair, mais il me semble naturel de sommer à partir de x_0. On aura $A = \sum_{i=0}^{N-1} 2^i = 2^N - 1$.
2. *Remarque :* L'algo revient juste à tirer au (pseudo-)hasard chaque bit du nombre à construire, en commençant par les poids faibles. En outre la suite à utiliser pour générer des nombres pseudo-aléatoires est précisée.
3. Pour `premier_rapide` : la question antéprécédente montre que pour tout $N \in \mathbb{N}$, l'appel `bbs(N)` renvoie un entier aléatoire entre 0 et $2^N - 1$. Ce n'est pas très pratique ici car `nb_max` n'a selon l'énoncé aucune raison d'être une puissance de deux.

   Cependant, `mystere(nb_max,2)` permet de récupérer le plus grand entier N tel que $2^N \leqslant$ `nb_max`. Ainsi, `bbs(mystere(nb_max,2))` renvoie bien un nombre entier strictement inférieur à N. Mais tous les entiers entre 2^N et `nb_max` $- 1$ seront ignorés.

   Le sujet aurait peut-être mieux fait d'imposer que `nb_max` soit une puissance de deux, ou d'utiliser le nombre de bits comme paramètre.

   Aure difficulté : `bbs` pourrait renvoyer 0 comme valeur de `p` auquel cas le calcul de `a**(p-1)` échoue, ou alors 1 auquel cas `a**(p-1)%p` vaut 1 sant que $p$soit premier. Il faut donc éliminer ces deux cas.

   Enfin, le test de Fermat ne fonctionne pas pour $a = p$ (d'où la condition $a > p$ dans l'énoncé). En effet, $p^{p-1} \equiv 0 \not\equiv 1[p]$. Ainsi par exemple le nombre 7 sera jugé non premier par ce test et sera donc écarté. C'est la raison pour laquelle l'énoncé demande que `nb_max` $\geqslant 12$: ainsi il y aura au moins un nombre premier qui soit > 7 et $<$ `nb_max` (c'est 11).

   Mais alors nouvelle difficulté : par exemple pour `nb_max` $= 12$, on aura `mystere(nb`$_{\texttt{m}}$`ax, 2)` $= 3$ donc les nombres seront tirés au hasard par `bbs` dans $[|0, 7|]$, ils seront alors tous jugés non premier par notre algo. Au final il faudra prendre `nb_max` $\geqslant 16$ pour que `mystere(nb`$_{\texttt{m}}$`ax, 2)` renvoie 4 et que l'algo puisse terminer.

4 Compter les nombres premiers

4.1 Calcul de $\pi(n)$ via un crible

4.2 Calcul approché via une intégrale généralisée

4.2.1 Estimation de li par quadrature numérique

1. Nous voudrons obtenir des résultats les plus précis possibles, ce qui sera obtenu lorsque `pas`tend vers 0. Ainsi, il est pertinent d'exprimer la complexité en fonction de `pas`, lorsque celui-ci tend vers 0. En outre, nous voudrons également prendre x grand, car pour des petiter valeurs, la valeur exacte de $\pi(n)$ peut être calculée. En résumé, nous allons exprimer la complexité en fonction de x et `pas`, lorsque le premier tend vers ∞ et le second vers 0.

   L'opération élémentaire la plus couteuse est sans doute l'appel à la fonction à intégrer, qui est en $O(1)$ d'après l'énoncé. Il y a un tel appel pour chaque rectangle de la subdivision, et il y a $\frac{x}{\texttt{pas}}$ rectangles.

   D'où une conplexité en $O\left(\frac{x}{\texttt{pas}}\right)$.
2. Les méthodes des rectangles centrés et des trapèzes ont la même complexité.

4.2.2 Analyse des résultats de `li_d`

1. L'écart relatif présente une asymptote en une valeur proche de 1.4 tout simplement car son calcul fait intervenir une division par `li_ref` qui s'annule (visible sur la figure 1).

2. Voici mon hypothèse et quelques commentaires concernant cette question :
   - Tout d'abord, à partir du moment où on n'est pas en train d'intégrer une fonction continue (par morceaux) sur un segment, il n'y a aucune raison que la méthode des rectangles converge.
     Je précise d'ailleurs qu'aucun théorème précis de convergence n'est cité au programme, je précise également que la notion d'intégrale impropre où la singularité est à l'intérieur de l'intervalle d'intégration n'est pas au programme de math, même de MP.
   - Du fait que $\ln(1-\varepsilon)$ et $-\ln(1+\varepsilon)$ sont égaux au premier ordre lorsque $\varepsilon \to 0$, on déduit que $\lim_{\varepsilon\to 0} \int_{1-\varepsilon}^{1+\varepsilon} \frac{1}{\ln(1+t)}\mathrm{dt} = 0$ (le calcul est laissé au lecteur, utiliser Taylor-Lagrange).
     Plus précisément, $\displaystyle\int_{1-\varepsilon}^{1+\varepsilon} \frac{1}{\ln(1+t)}\mathrm{dt} = O_{\varepsilon\to 0}(\varepsilon)$.
   - Voici alors une manière de corriger le calcul. On commence par fixer ε. Les intervalles $[0, 1-\varepsilon]$ et $[1+\varepsilon, x]$ sont de braves segments sur lesquels la fonction $x \mapsto \frac{1}{\ln x}$ est définie et continue. Donc la méthode des rectangles converge sur ces intervalles, et il existe un pas tel que l'erreur soit inférieure à ε. Comme en outre, l'erreur commise en supprimant $[1-\varepsilon, 1+\varepsilon]$ de l'intervalle d'intégration est aussi de l'ordre de ε, nous aurons une erreur totale en $O(\varepsilon)$.
     Donc en supprimant la condition `pas` $= \varepsilon$, nous pouvons obtenir un résultat correct. Dans le corrigé, j'ai pris un pas égal à ε^2, et le calcul semble correct.

4.3 Estimation de *li* via *Ei*

1. Pour obtenir une complexité en $O($`MAXIT`$)$, il faut penser à calculer les factorielles et les puissances de x en les gardant en mémoire au fur et à mesure.
   L'écriture propre des invariants de boucle peut vous sauver dans ce genre de fonction...

5 Évaluation des performances (BDD)

1. Plusieurs enregistrements ont la même valeur pour le champ `nom`.
2. (a) Nombre d'ordinateurs disponibles et quantité moyenne de mémoire vive.

```
SELECT COUNT(*) AS nb_ordi, AVG(RAM) AS RAM_moyenne
      FROM ordinateurs

```

   (b) Noms des PC sur lesquels l'algorithme `rectangles` n'a pas été testé pour la fonction `li`.

```
SELECT nom FROM ordinateurs
  WHERE nom NOT IN (
    SELECT teste_sur FROM fonctions
    WHERE algorithme="rectangles" and nom = "li"
  )

```

   Ou alors utiliser `EXCEPT` :

```
SELECT nom FROM ordinateurs
EXCEPT
SELECT teste_sur FROM fonction
      WHERE  algorithme="rectangles" and nom = "li"

```

   (c) Pour chaque test de `Ei` garder le nom de l'algo, du pc, et sa puissance. Trier du plus lent au plus rapide.

```
SELECT algorithme, teste_sur, ram, gflops
  FROM fonctions JOIN ordinateurs ON fonctions.teste_sur =
   ↪ ordinateurs.nom
  WHERE fonctions.nom = "Ei"
  ORDER BY temps_exec DESC

```