TP d'informatique n°7 : algorithmes dichotomiques

PCSI 2023 – 2024

Objectifs : Le but de ce TP est de mettre en évidence « expérimentalement » que pour résoudre certains problèmes, il est possible de trouver des algorithmes beaucoup plus efficaces que les algorithmes « naïfs » auxquels on pense dans un premier temps.

I À faire <u>avant</u> le tp (peu de travail)

Avant de venir en TP, lire les paragraphes II.1. , III.1. et III.3. Il n'est rien demandé de coder.

II Exponentiation

1. Description

Le but de cette partie est de construire une fonction `puissance(x:int, n:int)->int` calculant x^n ($n \geqslant 0$) sans s'autoriser l'opération `**` de python, mais seulement `+, -, *, //` (qui sont des opérations plus élémentaires au niveau du processeur).
Pour pouvoir manipuler de grands exposants sans dépasser la limite [1] de 64 bits, nous calculerons tous les résultats modulo un entier (opérateur %). Ainsi notre fonction sera plutôt `puissance(x:int, n:int, maxi:int)->int` pour calculer x^n mod $maxi$. Ce genre d'exponentiation modulaire avec de gros exposants n est en particulier utile en cryptographie il me semble (sécurisation des échanges avec https, wifi sécurisé, échange bancaire, etc...) et possède donc de réelles applications pratiques. Si vous voulez savoir à quoi sert l'exponentiation modulo un nombre, cherchez des informations sur l'algorithme RSA.

Remarque : il est important de calculer le modulo à chaque étape de calcul et non pas « une seule fois à la fin » pour éviter que les nombres calculés même dans les étapes intermédiaire reste suffisamment petits pour être manipuler sans problème par l'ordinateur. Avec certains des tests que nous allons faire, si on ne met pas le modulo, alors alors faudrait plusieurs centaines de Mo juste pour stocker un nombre, alors qu'un nombre entier est normalement codé sur 64 octets.

2. Algorithme naïf

Exercice n°1 En utilisant la définition de $x^n[maxi] = \underbrace{x \times x \times \cdots \times x}_{n \text{ fois}}[maxi]$, programmez une fonction `puissanceNaif(x:int, n:int, maxi:int)->int`.
Si vous le souhaitez, dans un premier temps, coder la puissance sans tenir compte du modulo, puis améliorer votre programme en en tenant compte.

Exercice n°2 Si vous ne l'avez pas déjà fait (vous auriez dû le faire !), testez votre fonction, entre autre avec $n = 0$. Testez avec plusieurs valeurs, vous pouvez vérifier avec la fonction puissance de python si vos exposants sont « suffisamment petit ». Par exemple `puissanceNaif(3, 1000, 2**31-1)->1651151508`.

1. Cf cours du second semestre. Cette limite n'est pas une limite indépassable en python, mais elle impacterait fortement la vitesse de calcul et fausserai l'exploitation des résultats.

Exercice n°3 En admettant qu'un passage dans votre boucle s'exécute en environ $0{,}1$ μs, combien de temps faudrait-il pour exécuter `puissanceNaif(3, 10**8, 2**31-1)`. Vérifier l'ordre de grandeur en faisant l'expérience. La valeur renvoyé doit être `puissanceNaif(3, 10**8, 2**31-1) ->1447121625`, mais ici, c'est surtout le temps d'exécution qui nous intéresse.

3. Algorithme rapide

La base de l'algorithme rapide est : si l'exposant n est pair, alors on peut utiliser l'égalité suivante : $x^n = x_2^{n_2}$ avec $x_2 = x \times x$ et $n_2 = n/2$ (division entière).

Exercice n°4 En utilisant l'algorithme naïf, combien de multiplication faudrait-il effectuer pour le calcul de $x_2^{n_2}$ par rapport à celui de x^n ?

Mais si le nouvel exposant est pair, on peut recommencer cette opération et re-diviser encore par 2 le nombre de multiplication à effectuer ! Et ainsi de suite. Lorsque l'exposant est impair, on fait une multiplication simple et l'exposant restant à calculer est pair.

L'algorithme complet est l'algorithme 1 page 2.

Algorithme 1 : Exponentiation rapide

Données : x un réel ou un entier et n un entier positif.
début
  $res \leftarrow 1$;
  tant que $n > 0$ **faire**
    si n *est impair* **alors**
      $res \leftarrow res * x$;
      $n \leftarrow n - 1$
    $x \leftarrow x * x$;
    $n \leftarrow n/2$;
  renvoie res;

Exercice n°5 Traduire cet algorithme en une fonction python `puissanceRap(x:int, n:int, maxi :int) ->int`. On prendra soin d'utiliser la division entière lors de la division par 2 pour manipuler des nombres entiers (4/2 N'est PAS un entier en python). Si vous le souhaitez, ne rajoutez le modulo qu'après (par contre, il est important de bien implémenter le modulo avant d'essayer avec de grands exposants pour éviter de « planter » l'ordinateur, et de l'ajouter à chaque calcul sur x ou sur res).

Exercice n°6 Si vous ne l'avez pas fait (vous avez tort !), testez votre fonction avec les mêmes valeurs qu'au paragraphe précédent.

Exercice n°7 En admettant qu'un passage dans votre boucle s'exécute en environ $0{,}1$ μs, combien de temps faudrait-il pour exécuter `puissanceRap(3, 2**27, 2**31-1)`. Si possible, vérifier l'ordre de grandeur en faisant l'expérience.

Exercice n°8 Comparer les ordres de grandeurs de 2^{27} et 10^8. Sur cet exemple de $3^{(10^8)}$, la puissance rapide est elle plus rapide ? De combien de fois environ ?

III Recherche d'un élément dans un tableau trié

1. Position du problème

On considère une liste L d'objets triés [2], c'est-à-dire tel que chaque élément est supérieur ou égal à l'élément précédent : $L[i+1] > L[i]$. Par la suite, dans l'énoncé, on notera n le nombre d'élément de cette liste L. On veut coder une fonction `recherche(x:"objet", L:list)->bool` qui renvoie `True` si x est présent dans la liste L et $False$ sinon.

2. Algorithme naïf

Exercice n°9 Sans utiliser le fait que la liste L est triée et en parcourant simplement la liste comme dans le TP01, écrire une fonction `appartientNaif(x, L)` .

Exercice n°10 Si vous ne l'avez pas fait (vous auriez dû !), testez votre fonction. Par exemple `rechercheNaif(4,[1, 2, 3])-> False` et `rechercheNaif("c",list("abc"))-> True`.

Exercice n°11 Si la longueur de la liste double, et que $x \notin L$, comment évolue (approximativement) le temps de calcul de votre fonction ?
(a) reste constant (b) double (c) quadruple (d) augmente, mais moins que $\times 2$?

3. Recherche par dichotomie

Regardez le diaporama dans le lien ci-dessous/ci-contre, **OU** lisez le paragraphe en dessous du lien.
`http://grenard.dyndns.org/diaporama/rechercheDicho.pdf`
Principe : on va utiliser le fait que la liste est triée pour éliminer rapidement une grande partie de la liste. Imaginez la différence entre chercher dans un dictionnaire « normal » et un dictionnaire dans lequel les mots auraient été mélangés ! Pour cela :

- on regarde au milieu de la liste l'élément présent ;
- en fonction de sa « valeur » par rapport à ce que l'on cherche, soit on a trouvé ce qui nous intéresse et on renvoie $True$, soit on continue de chercher à gauche ou à droite ;
- on recommence en cherchant au milieu de la nouvelle partie pertinente.

Remarque : Cet algorithme est important et vous devez savoir le coder sans erreur. Nous étudierons son temps de calcul en détail en cours.

Exercice n°12 En supposant que la « partie pertinente » soit comprise entre **les indices** deb et fin. Quel est l'indice du milieu ? Testez votre formule avec $deb = 2$ et $fin = 4$, puis avec $deb = 2$ et $fin = 5$

Exercice n°13 Appliquer l'algorithme à la main pour chercher $x = 5$ dans $L = [-3, -3, -3, -2, -1, 0, 2, 3, 5, 7, 10, 12, 15, 18]$.

Exercice n°14 Appliquer l'algorithme à la main pour chercher $x = 4$ dans la même liste que ci-dessus. Quant arrête-t-on l'algorithme lorsque $x \notin L$? En déduire la condition de boucle.

2. il s'agit donc d'objet pour lesquels on peut utiliser les opérateurs $<, >, >=, <=$.

Exercice n°15 Écrire une fonction python `rechercheDicho(x, L)` réalisant la recherche de x dans L par l'algorithme de recherche dichotomique en supposant L trié (on ne le vérifie pas avant).

Exercice n°16 Testez votre fonction dans les cas suivant :
- `rechercheDicho(4,[1, 2, 3])-> False` et `rechercheDicho("c",list("abc")) -> True`.
- $x \in L$ (doit renvoyer vrai bien sûr), avec un nombre d'éléments pair, puis impair ;
- $x \notin L$ et $min(L) < x < max(L)$;
- $x \in L$ et $min(L) = x$;
- $x \in L$ et $max(L) = x$;
- $x \notin L$ et $min(L) > x$;
- $x \notin L$ et $x > max(L)$;

Exercice n°17 Si la longueur de la liste double, et que $x \notin L$, comment évolue (approximativement) le temps de calcul de votre fonction ?
(a) reste constant (b) double (c) quadruple (d) augmente, mais moins que $\times 2$?
Pourquoi ? Combien de passage en plus dans la boucle fait-on ?

IV Si le TP est trop court

Exercice n°18 En utilisant le TP sur matplotlib, ainsi que la fonction perfcounteur du module time, mesurez le temps d'exécution de vos programmes `puissanceRap` et `puissanceNaif` en fonction de n (valeur de l'exposant) pour tracer l'évolution du temps de calcul avec n (ou éventuellement en échelle semi-logarithmique si vos calculs montre qu'un logarithme doit intervenir).

V À faire pour la prochaine séance

Exercice n°19 Sans regardez vos notes (sauf pour vérifier), reprogrammez la fonction `rechercheNaif`

Exercice n°20 Sans regardez vos notes (sauf pour vérifier), reprogrammez la fonction `rechercheDicho`

Table des matières