

## TP5 : Puissance rapide, Code cryptographique RSA

N'oubliez pas d'exécuter (valider avec la touche Entrée) les commandes Maple (texte en rouge) avant de les utiliser.

### – Puissance rapide

On a besoin de calculer des puissances  $x^n \bmod m$  avec  $x, n, m$  entiers de grande taille. On va comparer la vitesse de l'algorithme naïf et celle de l'algorithme rapide de calcul de puissances expliqué en cours.

Exercice 1 : Ecrire une fonction qui rende  $x^n$  en utilisant l'algorithme naïf par multiplications successives et une fonction qui rende  $x^n$  en utilisant l'algorithme rapide. Chercher des entiers tels que les deux algorithmes aient des temps de calcul significativement différents.

En faisant varier  $n$  de manière exponentielle comparer graphiquement la vitesse de l'algorithme naïf et celle de l'algorithme rapide.

Que se passe-t-il si  $n$  est très grand (de l'ordre de  $10^{50}$ ) ?

### – Solution

L'algorithme naïf :

```
[ > PuissNaif:=proc(x,n)
  local i,res;
  res:=1;
  for i from 1 to n do res:=res*x od;
  res
end;
```

On initialise res à 1 et non à x pour que cette fonction marche quand  $n = 0$ .

```
[ > PuissNaif(11,3),11^3,PuissNaif(11,0);
      1331, 1331, 1
```

L'algorithme rapide :

```
[ > PuissRap:=proc(x,n)
  if n=0 then 1
  elif irem(n,2)=0 then PuissRap(x*x,iquo(n,2)) else
  x*PuissRap(x*x,iquo(n-1,2)) fi
end:
[ > PuissRap(11,3),11^3,PuissRap(11,0);
      1331, 1331, 1
```

On calcule les temps d'exécution sur des exemples assez grands :

```
[ > t:=time():PuissNaif(11,3^9):time()-t;
      4.329
[ > t:=time():PuissRap(11,3^9):time()-t;
      0.234
```

Les temps sont significativement différents.

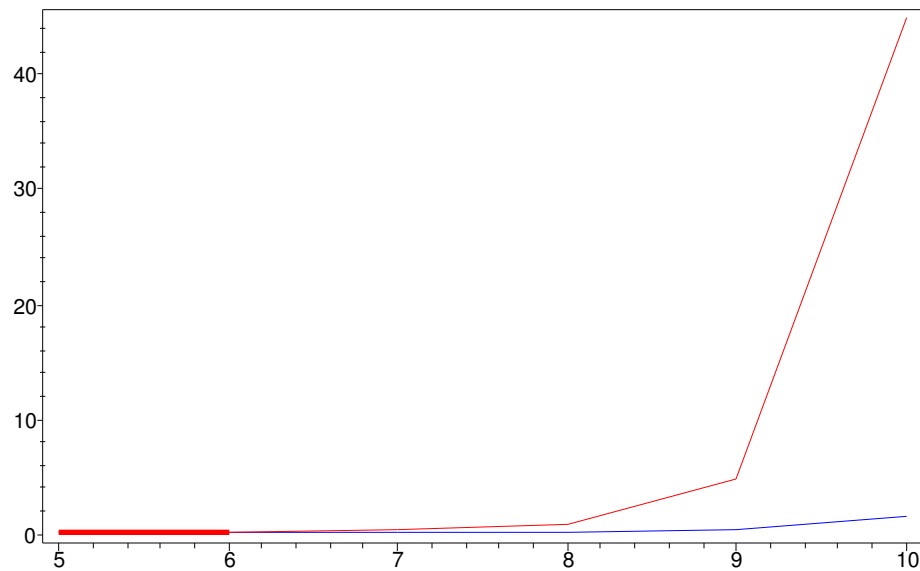
On a choisi de calculer  $11^{19683}$  pour éviter d'éventuelles particularités des bases 2 et 10.

On fait une série de mesures pour  $11^{(3^i)}$  :

```
[ > s1:=NULL: for i from 5 to 10 do t:=time():
  PuissNaif(11,3^i): t:=time()-t: s1:=s1,[i,t] od: s1;
      [5, 0], [6, 0.015], [7, 0.047], [8, 0.531], [9, 4.672], [10, 44.641]
[ > s2:=NULL: for i from 5 to 10 do t:=time():
  PuissRap(11,3^i): t:=time()-t: s2:=s2,[i,t] od: s2;
      [5, 0], [6, 0], [7, 0.015], [8, 0.016], [9, 0.219], [10, 1.375]
```

et les graphiques :

```
[ > plot([[s1],[s2]],color=[red,blue]);
```



On voit que les temps de l'algorithme naïf décollent sévèrement par rapport à l'algorithme rapide, mais pourquoi les temps de l'algorithme rapide n'ont-ils pas tout à fait l'air d'une droite ( $\log_2(3^i)$ ) ?

La taille des nombres compte aussi dans la vitesse et d'ailleurs si les nombres sont trop grands :

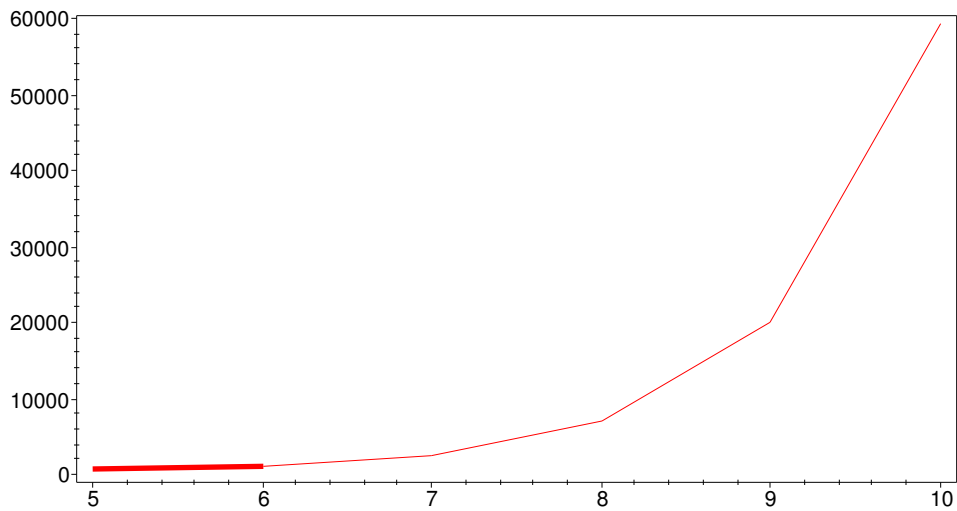
```
[ > PuissRap(11,3^100);
[ Error, (in PuissRap) object too large
```

Maple ne peut plus faire le calcul à cause de la taille des entiers.

On va donc compter le nombre de multiplications pour chaque fonction et plus le temps de calcul.

Pour l'algorithme naïf c'est très simple : pour calculer  $x^n$  on fait  $n$  multiplications, par exemple  $11^{(3^{10})}$  demande  $3^{10} = 59049$  multiplications.

```
[ > s1:=NULL: for i from 5 to 10 do s1:=s1,[i,3^i] od: s1;
      [5, 243], [6, 729], [7, 2187], [8, 6561], [9, 19683], [10, 59049]
[ > plot([s1]);
```



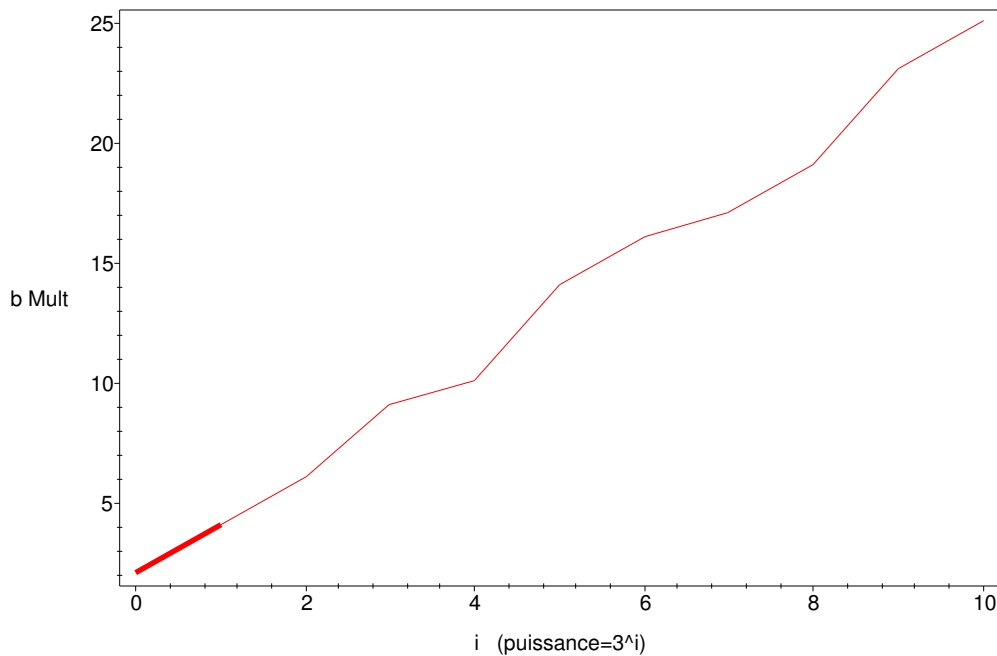
Pour la puissance rapide c'est plus compliqué, on crée une fonction de comptage et on démarre les comptes à  $3^0$

```
> CptPuissRap:=proc(n)
  if n=0 then 0
  elif irem(n,2)=0 then CptPuissRap(iquo(n,2))+1 else
  CptPuissRap(iquo(n-1,2))+2 fi
end:

> s2:=NULL: for i from 0 to 10 do
  s2:=s2,[i,CptPuissRap(3^i)] od: s2;
  [0, 2], [1, 4], [2, 6], [3, 9], [4, 10], [5, 14], [6, 16], [7, 17], [8, 19], [9, 23], [10, 25]
```

On constate par exemple que  $11^{(3^{10})}$  ne demande que 25 multiplications au lieu de  $3^{10} = 59049$  !

```
> plot([s2],labels=['i (puissance=3^i)', 'Nb Mult']);
```



On a bien à peu près une droite et sa pente de 2,3 correspond au nombre de multiplications compris entre  $\log_2(3^i) = \log_2(3) i$  et  $2 \log_2(3^i) = 2 \log_2(3) i$  (cas pair et cas impair).

```
> evalf(log[2](3)), 2*evalf(log[2](3));
1.584962501, 3.169925002
```

**Exercice 2 :** Ecrire une fonction qui rende  $x^n \bmod m$  en utilisant l'algorithme rapide.

Recommencer vos tests précédents en prenant  $m$  à 10 chiffres. Que constatez-vous ? Essayer avec des puissances plus grandes et comparer graphiquement avec la puissance modulaire de Maple &^.

### — Solution

```
> PuissRapMod:=proc(x,n,m)
  if n=0 then 1
  elif irem(n,2)=0 then PuissRapMod(x*x mod m,iquo(n,2),m)
  else x*PuissRapMod(x*x mod m,iquo(n-1,2),m) mod m fi
end;
```

On réduit modulo  $m$  à chaque étape du calcul.

```
> s3:=NULL: for i from 5 to 10 do t:=time():
  PuissRapMod(11,3^i,1234567890): t:=time()-t: s3:=s3,[i,t]
od: s3;
```

[5, 0], [6, 0], [7, 0], [8, 0], [9, 0], [10, 0]

Les temps ne sont pas mesurables, il faut tester avec des puissances beaucoup plus grandes.

```
> s4:=NULL: for i to 2001 by 50 do t:=time():
  PuissRapMod(111111111,3^i,1234567890): t:=time()-t:
  s4:=s4,[i,t] od: s4;
Error, (in PuissRapMod) too many levels of recursion
```

```
[1, 0], [51, 0], [101, 0.016], [151, 0], [201, 0], [251, 0], [301, 0.015], [351, 0.016],  
[401, 0.015]
```

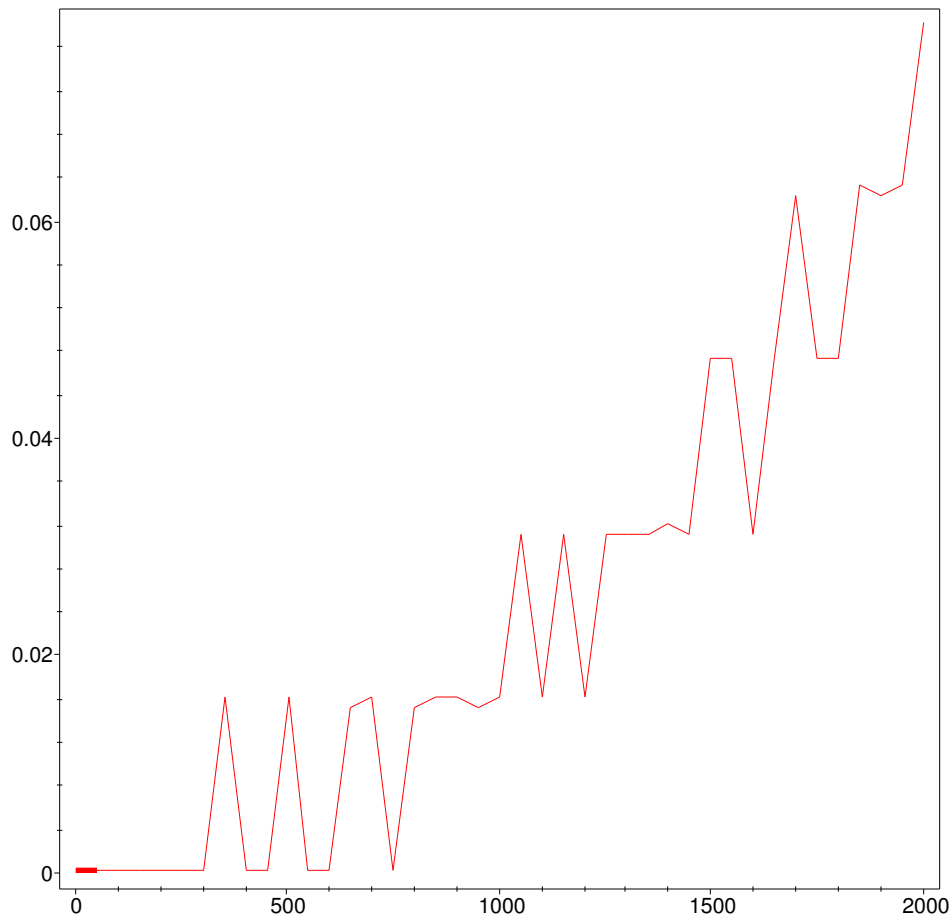
On n'arrive pas à faire de mesure : les temps sont quasi-nulles puis on atteint une limite du nombre de niveau d'appels récursifs acceptés par Maple.

On essaie avec &^

```
> s4:=NULL: for i to 2001 by 50 do t:=time():  
111111111&^(3^i) mod 1234567890: t:=time()-t: s4:=s4,[i,t]  
od: s4;
```

```
[1, 0], [51, 0], [101, 0], [151, 0], [201, 0], [251, 0], [301, 0], [351, 0.016], [401, 0],  
[451, 0], [501, 0.016], [551, 0], [601, 0], [651, 0.015], [701, 0.016], [751, 0],  
[801, 0.015], [851, 0.016], [901, 0.016], [951, 0.015], [1001, 0.016], [1051, 0.031],  
[1101, 0.016], [1151, 0.031], [1201, 0.016], [1251, 0.031], [1301, 0.031],  
[1351, 0.031], [1401, 0.032], [1451, 0.031], [1501, 0.047], [1551, 0.047],  
[1601, 0.031], [1651, 0.047], [1701, 0.062], [1751, 0.047], [1801, 0.047],  
[1851, 0.063], [1901, 0.062], [1951, 0.063], [2001, 0.078]
```

```
> plot([s4]);;
```



C'est plus ou moins une droite et on peut aller jusqu'à de très grandes puissances :

```
> evalf(3^2001);
```

0.5243613755 10<sup>955</sup>

## Code cryptographique RSA

Vous allez fabriquer un code RSA :

- à l'aide de **isprime** construire deux nombres premiers distincts à 20 chiffres et  $n$  leur produit
- calculez  $\phi(n)$ , un nombre  $c$  compris entre 1 et  $\phi(n) - 1$  premier avec  $\phi(n)$
- vous pouvez maintenant diffuser votre méthode de cryptage : si quelqu'un souhaite vous envoyer un message  $m$  (de moins de 20 chiffres) crypté vous lui donnez  $c$  et  $n$  et il vous envoie  $m_c = m^c \pmod n$ .
- pour décoder le message crypté il faut calculer l'inverse de  $c$  modulo  $\phi(n)$ . Décodez le message crypté  $m_c$ .

- quelqu'un qui n'a en possession que  $c$  et  $n$  (la méthode de cryptage) doit calculer  $\phi(n)$  pour pouvoir décrypter . Constatez que ça semble très long !

### – Solution

Une fonction pour introduire du hasard

```
[ > tire1:=rand(10^19..10^20):
```

On tire un point de départ :

```
[ > dep:=tire1();
```

```
dep := 60571674960498833103
```

On va parcourir les impairs jusqu'à tomber sur un nombre (très probablement) premier

```
[ > while not(isprime(dep)) do dep:=dep+2 od:p1:=dep;
```

```
p1 := 60571674960498833221
```

Remarquez la rapidité : il a fallu tester une bonne soixantaine de (gros) candidats.

On recommence pour construire p2

```
[ > dep:=tire1();
```

```
dep := 91395307920624940194
```

Il est pair, on lui rajoute 1

```
[ > dep:=dep+1;
```

```
dep := 91395307920624940195
```

```
[ > while not(isprime(dep)) do dep:=dep+2 od:p2:=dep;
```

```
p2 := 91395307920624940207
```

Et enfin  $n$

```
[ > n:=p1*p2;
```

```
n := 5535966884282798374122171267976390216747
```

D'où  $\phi(n)$

```
[ > phin:=(p1-1)*(p2-1);
```

```
phin := 5535966884282798373970204285095266443320
```

On tire  $c$  au hasard :

```
[ > tire2:=rand(1..phin-1):
```

```
[ > c:=tire2();
```

```
c := 1069333996470403175725398116693423050809
```

Est-il premier avec  $\phi(n)$  ?

```
[ > igcd(c,phin);
```

```
9
```

Non ? On en tire un autre jusqu'à ce qu'il soit premier avec  $\phi(n)$  :

```
[ > c:=tire2();igcd(c,phin);
```

```
c := 5028738945276122337583856046882237748469
```

```
1
```

Une fonction pour construire des messages au hasard :

```
[ > tire3:=rand(1..10^19):
```

Un message super secret !

```
[ > m:=tire3();
```

```
m := 7114594552921209929
```

Le message crypté

```
[ > mc:=m&^c mod n;
```

```
mc := 3926418805866016170086844275596515515591
```

Vous recevez  $m_c$

Vous devez calculer  $d$  l'inverse de  $c$  modulo  $\phi(n)$

```
[ > igcdex(c,phin,'d');
                                     1
[ > d;c*d mod phin;
                                     1
                                     990390118781583465028617377378275836389
```

Remarque : ce calcul n'est pas fait à chaque message mais une fois pour toute pour le code  $n, c$ .

On décrypte

```
[ > md:=mc&^d mod n;
                                     md := 7114594552921209929
[ > m-md;
                                     0
```

On retrouve bien le message d'origine.

Quelqu'un connaît  $c$  et  $n$  et veut décrypter  $m_c$

Il lui faut  $\phi(n)$

```
[ > with(numtheory):
[ Warning, new definition for order
[ > phi(n);
[ Warning, computation interrupted
[ > ifactor(n);
[ Warning, computation interrupted
```