Tri patience – écriture décimale de grands entiers

Quelques remarques générales

Le sujet est constitué de deux problèmes indépendants. Les fonctions demandées doivent être écrites dans le langage OCaml. On prendra bien soin à veiller à la lisibilité du code proposé, en choisissant judicieusement les noms de variables utilisés, et en assortissant les fonctions de commentaires ou d'explications brèves mais pertinentes permettant de comprendre les choix effectués.

Vous pouvez introduire toutes les fonctions auxiliaires dont vous avez besoin. Vous pouvez également utiliser dans une question toutes les fonctions décrites dans les questions précédentes du même problème, et ce *même si vous n'avez pas réussi à en proposer une implémentation.*

Si d'aventure vous trouvez ce que vous pensez être une erreur dans le sujet, indiquez-le sur votre copie, en précisant les choix que vous avez fait pour la contourner.

Précisons qu'à l'intérieur de chaque problème, les sections sont partiellement indépendantes, et une section peut être abordée sans que vous soyez nécessairement parvenu à traiter tout ou partie de la section précédente. En revanche, il sera utile de lire les parties précédentes pour avoir connaissance des objets manipulés et des types qui les définissent, ainsi que des fonctions dont vous pourriez avoir l'usage.

Fonctions OCaml utiles

Vous pouvez, dans ce devoi, utiliser librement toutes les fonctions ne provenant pas d'un module (fonctions ne contenant pas de « point » dans leur nom, plus les fonctions du module `List` (qui débutent par « `List.` »).

On rappelle ci-après quelques-unes des fonctions les plus utiles sur les listes. Bien que cela ne soit pas indispensable dans ce devoir, vous êtes libre d'utiliser d'autres fonctions si cela vous semble utile, qu'elles aient été vues en cours ou non, sous réserve de le faire avec soin et pertinence.

`List.length : 'a list -> int`

Renvoie le nombre d'éléments dans la liste fournie en argument. Cette fonction a une complexité $O(n)$ linéaire en la taille de la liste.

`List.hd : 'a list -> 'a`

Renvoie le premier élément de la liste fournie en argument. $O(1)$.

`List.tl : 'a list -> 'a`

Renvoie la liste fournie en argument privée de son premier élément. $O(1)$.

`List.rev : 'a list -> 'a list`

Retourne une nouvelle liste contenant les éléments de la liste fournie en argument en ordre inverse.

`List.mem : 'a -> 'a list -> bool`

« `List.mem x lst` » renvoie un booléen indiquant si au moins un élément de la liste `lst` est égal à `x`. Complexité dans le pire des cas linéaire en la taille de la liste ($O(n)$).

`List.iter : ('a -> unit) -> 'a list -> unit`

« `List.iter f lst` » exécute $f\,a_i$ successivement pour chacun des éléments a_i de la liste `lst` (dans l'ordre).

`List.map : ('a -> 'b) -> 'a list -> 'b list`

« `List.map f lst` » renvoie la liste `[f`a_0`; f`a_1`; ...; f`a_{n-1}`]` où les a_i sont les éléments de la liste `lst`. L'ordre d'évaluation des $f\,a_i$ n'est pas spécifiée.

1 Tri patience

1.1 Jeu de patience

Dans ce problème, nous allons nous intéresser à une méthode de tri originale, basée du le principe des jeux de cartes de type « patience ». Dans un tel jeu, les cartes sont posées sur la table en un ou plusieurs tas avec une règle simple : on ne peut poser une carte que sur une carte de valeur plus grande (on peut poser un 6 sur un 9, mais pas un 9 sur un 6). Attention, nous allons nous inspirer de ce jeu mais les détails pourront être légèrement différents de la variante de jeu de cartes que vous pourriez connaître !

On souhaite trier des objets, par exemple des entiers [1], pouvant être comparés avec les opérateurs de comparaison usuels tels que `<=`. Pour y parvenir, dans un premier temps, nous allons considérer les éléments un par un et les placer selon des règles inspirées du jeu de patience :

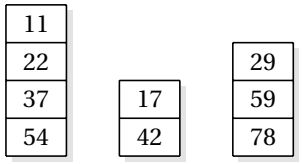
- initialement, il n'y a aucun tas ;
- lorsque l'on considère un élément x, deux cas peuvent se présenter :
  - si x est **inférieur ou égal** à au moins un des éléments au sommet d'un des tas existants, alors on place x au-dessus du tas le plus à gauche qui remplit cette condition ;
  - si x est **strictement plus grand** que l'ensemble des éléments au sommet de chacun des tas, alors on crée un nouveau tas, à droite des tas existants, dans

1. Dans la suite, on supposera dans les fonctions que les objets sont de type `'a`, mais les exemples utiliseront des entiers

lesquels on place x (en particulier, le premier élément sera nécessairement placé sur un nouveau tas).

Prenons un exemple : on suppose qu'il y a, à un instant donné, trois tas. De gauche à droite :

- un premier tas, dont les cartes de bas en haut sont 54, 37, 22 et 11 ;
- un second tas, dont les cartes de bas en haut sont 42 et 17 ;
- un troisième tas, dont les cartes de bas en haut sont 78, 59 et 29.



Si l'élément suivant considéré était $x = 7$, il serait plus petit que les éléments 11, 17 et 29 au sommet de chacun des tas. On pourrait donc le placer au-dessus de n'importe lequel de ces tas, et on choisirait alors le plus à gauche : il se placerait au-dessus du 11.

Si c'était $x = 14$, il est plus grand que 11 mais plus petit que 17 et 29, donc il faudrait le placer sur le tas central, par-dessus le 17.

Si c'était $x = 31$, il est plus grand que les éléments au sommet de chacun des tas, il conviendrait alors de créer un quatrième tas tout à droite pour y placer cet élément 31.

1. On suppose qu'initialement il n'y a aucun tas, et on considère les dix éléments suivants, dans cet ordre : 5, 3, 7, 4, 2, 9, 6, 8, 0 et 1 (le 5 étant donc le premier élément placé, dans un tout nouveau premier tas). Dessiner le résultat obtenu si l'on suit les règles énoncées précédemment.

Pour représenter l'état du « jeu » à un instant donné, on utilise une liste de listes (non vides) d'éléments. Chaque liste à l'intérieur de la liste de liste représente un tas, les tas étant considérés de gauche à droite. L'élément en tête de ces listes (à gauche) est le sommet du tas, celui à droite le fond du tas. L'état du jeu tel que décrit précédemment est donc représenté par la liste de listes :

```
[ [11; 22; 37; 54]; [17; 42]; [29; 59; 78] ]
```

2. Proposer une fonction `add` de type `'a -> 'a list list -> 'a list list` qui prend un élément x et une liste de listes représentant l'état du jeu, et retourne une liste de listes représentant l'état du jeu après avoir posé x selon les règles. Par exemple, en accord avec les explications précédentes,

- « `add 7 [ [11; 22; 37; 54]; [17; 42]; [29; 59; 78] ]` » devra renvoyer la liste de listes `[ [7; 11; 22; 37; 54]; [17; 42]; [29; 59; 78] ]`
- « `add 14 [ [11; 22; 37; 54]; [17; 42]; [29; 59; 78] ]` » devra renvoyer la liste de listes `[ [11; 22; 37; 54]; [14; 17; 42]; [29; 59; 78] ]`
- « `add 31 [ [11; 22; 37; 54]; [17; 42]; [29; 59; 78] ]` » devra renvoyer la liste de listes `[ [11; 22; 37; 54]; [17; 42]; [29; 59; 78]; [31] ]`

3. En déduire une fonction `play` de type `'a list -> 'a list list` prenant en argument une liste $[x_0; x_1; x_2; \ldots; x_{n-1}]$ de n éléments x_i, plaçant ces éléments un par un suivant les règles précédentes, en considérant ces éléments de gauche à droite (d'abord x_0, puis x_1, etc.) et renvoyant la liste de listes correspondant à l'état du jeu une fois tous les éléments posés.

4. Justifier soigneusement que, si l'on part d'aucun tas et que l'on suit les règles, à tout moment, les éléments au sommet des tas, pris de la gauche vers la droite (11, 17 et 29 sur l'exemple initial) sont toujours rangés dans un ordre croissant. Est-ce un ordre strictement croissant ou croissant au sens large ?

5. Toujours en supposant que l'on part d'aucun tas et que l'on place les n éléments, un par un, combien de tas peut-il y avoir à la fin ? Proposer un ordre pour n éléments x_i donnant le nombre minimal de tas, et un ordre pour n éléments x_i donnant le nombre maximal de tas.

Note : on peut montrer que l'espérance du nombre de tas une fois tous les éléments posés, si les éléments sont tous distincts et dans un ordre parfaitement aléatoire, est $\sqrt{n}$.

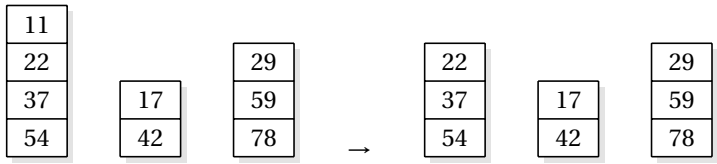
6. Quelle est la complexité temporelle dans le pire des cas pour `play` pour une liste à n éléments ? La complexité dans le meilleur des cas ?

1.2 Obtenir un tri

À l'issue de la patience, nous avons déjà montré que le plus petit élément se trouvait au sommet du tas le plus à gauche. Si l'on veut trier les éléments par ordre croissant, cet élément devra se trouver en première position de la liste triée.

7. Proposer une fonction `smallest` de signature `'a list list -> 'a` prenant en argument une liste de listes représentant les tas (on suppose qu'il y a au moins un tas, et qu'aucun tas n'est vide) et retourne l'élément situé au sommet du tas le plus à gauche.

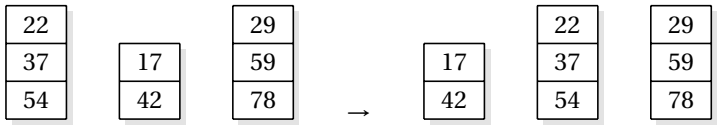
Pour la suite, il nous faudra enlever cet élément situé au sommet de la première pile. Cependant, aucun tas ne doit jamais être vide. Si l'élément retiré était l'unique élément du tas de gauche, on décale chacun des autres tas d'un cran vers la gauche pour ne pas laisser de « trou ». Sinon, les tas restent en place :



8. Proposer une fonction `remove` de signature `'a list list -> 'a list list` prenant en argument une liste de listes représentant l'ensemble des tas, retirant l'élément au sommet du tas le plus à gauche et renvoyant la liste de listes décrivant l'état après le

retrait. La liste de listes ainsi renvoyée ne doit pas contenir de liste vide (la liste fournie n'en contenait pas).

Bien évidemment, après avoir retiré l'élément au sommet du tas le plus à gauche, on n'a plus la garantie que les éléments aux sommets de chacun des tas demeurent rangés par ordre croissant. On souhaite rétablir cette propriété. Pour ce faire, on va prendre le tas le plus à gauche, et opérer une insertion de ce tas dans la liste des autres tas (dont les éléments au sommets sont encore rangés par ordre croissant), de façon identique à une insertion dans un tri par insertion. On ne comparera les tas qu'en fonction de l'élément à leur sommet (on rappelle qu'aucun tas n'est vide).



9. Coder une fonction `insert` de signature `'a list -> 'a list list -> 'a list list` prenant en argument un tas et une liste de tas (triés par ordre croissant des éléments à leur sommet), et insérant le premier tas dans la liste des tas en un temps linéaire en le nombre de tas. Par exemple, comme illustré ci-dessus, « `insert [22; 37; 54] [ [17; 42]; [29; 59; 78] ]` » devra renvoyer `[ [17; 42]; [22; 37; 54]; [29; 59; 78] ]`

Après un appel à la fonction `remove` suivi d'un appel à la fonction `insert`, on retrouve tous les éléments, à l'exception du plus petit élément que l'on a extrait, dans un ensemble de tas dont les sommets sont triés par ordre croissant. En particulier, le second plus petit élément se trouve à présent au sommet du tas le plus à gauche.

10. En déduire une fonction `sort` de signature `'a list -> 'a list` qui utilise les éléments précédents (`play`, `smallest`, `remove`, `insert`) afin de prendre en argument une liste et de retourner une liste dans laquelle les éléments ont été triés par ordre croissant.

11. Quelle est la complexité, dans le pire cas, de la fonction `sort` ?

12. Le tri est-il stable ? On justifiera la réponse.

1.3 Sous-séquences strictement croissantes

Le principe de la patience permet d'obtenir un autre résultat intéressant sur les listes.

Une *sous-séquence strictement croissante* de longueur p d'une liste $[x_0; x_1; x_2; \ldots; x_{n-1}]$ est une liste $\left[x_{\phi(0)}; x_{\phi(1)}; \ldots; x_{\phi(p-1)}\right]$ où ϕ est une fonction croissante de $[\![0\,..\,p-1]\!]$ dans $[\![0\,..\,n-1]\!]$ vérifiant, pour tout $0 < i < p$, $x_{\phi(i)} > x_{(i-1)}$.

Par exemple, pour la liste $[5;3;7;4;2;9;6;8;0;1]$, la fonction ϕ de $[\![0\,..\,2]\!]$ dans $[\![0\,..\,9]\!]$ définie par $\phi(0) = 1$, $\phi(1) = 3$ et $\phi(2) = 7$ définit la sous-séquence croissante $[3;4;8]$.

Une sous-séquence strictement croissante d'une liste donnée est dite de longueur maximale s'il n'existe pas, pour la même liste, de sous-séquence strictement croissante de longueur strictement plus grande.

13. Montrer que le la longueur des sous-séquences strictement croissantes de longueur maximale est liée au nombre de tas obtenus lorsque l'on a appliqué, sur la liste, les règles précédentes.

14. En déduire la longueur maximale des sous-séquences strictement croissantes pour la liste $[5;3;7;4;2;9;6;8;0;1]$, et exhiber une sous-séquence strictement croissante de longueur maximale.

15. Expliquer précisément comment, à partir de l'ensemble des tas obtenu avec la patience, on peut construire une telle sous-séquence croissante de longueur maximale.

16. Proposer une fonction `long_incr_subseq` de signature `'a list -> 'a list`, prenant en argument une liste d'éléments et retournant une sous-séquence strictement croissante de longueur maximale, en utilisant la fonction `play` et les idées exposées à la question précédente.

2 Grands entiers

2.1 Introduction

Il est fréquent de croiser des problèmes de mathématiques faisant intervenir de (très) grands entiers. Par exemple, on peut vouloir savoir combien de zéros se trouvent à l'extrémité droite de l'écriture décimale de 2024! (ou ! représente la fonction factorielle), ou bien combien de fois le chiffre 7 s'y trouve.

On peut parfois y répondre avec des critères arithmétiques, mais de temps en temps, il faudra pouvoir écrire le nombre en entier. Bien évidemment, il n'est pas question d'utiliser des entiers OCaml pour les calculs car ces nombres sont bien plus grands que `max_int` !

Considérons la première question : combien de zéros se trouvent à droite de l'écriture décimale de 2024!? Pour illustrer le problème, prenons le cas plus simple de 42!. Son écriture décimale est :

$$1405006117752879898543142606244511569936384000000000$$

Le nombre de zéros recherché est donc 9.

Soit n un entier strictement positif, et d un entier strictement positif. Il existe un unique couple d'entiers naturels (i, p) tel que $n = d^i \times p$ où p ne divise pas n.

Par exemple, pour $n = 42$ et $d = 3$, on a $i = 1$ et $r = 14$ ($42 = 3^1 \times 14$). Pour $n = 54$ et $d = 3$, on a $i = 3$ et $p = 2$ ($54 = 3^3 \times 2$).

1. Proposer une fonction `exponent` de signature `int -> int -> int` telle que « `exponent d n` » renvoie `i`, tel que défini précédemment.

On définit une fonction récursive `foo` par :

```
let rec foo d = function
  | 0 -> 0
  | n -> exponent d n + foo d (n-1)
```

2. Préciser la signature de `foo`.

3. Justifier que l'on peut, à partir de cette fonction `foo`, construire une fonction `nb_zeros` de signature `int -> int` prenant en argument un entier $n > 0$ et renvoyant le nombre de zéros à droite de l'écriture décimale de n, et proposer une telle fonction.

4. Cette fonction est relativement coûteuse en terme de calculs. Proposer une autre fonction `nb_zeros_bis` réalisant la même opération mais avec une complexité moindre.

2.2 Grands entiers

Malheureusement, il n'est pas toujours possible d'utiliser des raisonnements arithmétique pour parvenir aux résultats. Dans la suite, nous allons mettre en place une solution pour manipuler des entiers de taille arbitraire (possiblement bien plus grands que `int_max`). Il est possible, pour ce faire, d'utiliser des chaînes de caractères. Nous allons utiliser ici une approche différente.

Soit $m = 10^c$ une puissance de dix (avec c entier strictement positif). Pour tout entier $n > 0$, on peut trouver un *unique* ensemble de p coefficients $(a_i)_{i \in [\![0..p-1]\!]}$ vérifiant :

- $n = \sum_{k=0}^{p-1} a_i \mathrm{M}^i = \mathrm{a}_{p-1} \times m^{p-1} + \ldots + a_2 \times m^2 + a_1 \times m + a_0$;
- pour tout $0 \leqslant i < p$, $a_i \in [\![0\,..\,m-1]\!]$;
- $a_{p-1} \neq 0$.

Par exemple, si $m = 100$ ($c = 2$), l'entier $n = 1234567$ s'écrit $1 \times m^3 + 23 \times m^2 + 45 \times m + 67$.

Ce qui donne $[a_0 = 67; a_1 = 45; a_2 = 23; a_3 = 1]$.

Pour représenter un tel entier n en OCaml, on utilisera donc une liste de coefficients a_i, rangés par i croissants. L'entier n sera ainsi représenté par la liste OCaml `[67; 45; 23; 1]`. Comme il n'y a pas de limitations quant à la taille des listes en OCaml (hors capacité mémoire), il n'y a pas de limite à la taille des entiers que l'on pourra ainsi manipuler. L'entier 42! présenté tantôt par exemple sera représenté (toujours si $m = 100$) par

```
[00; 00; 00; 40; 38; 36; 99; 56; 11; 45; 24; 06; 26; 14; 43; 85;
                                 89; 79; 28; 75; 17; 61; 00; 05; 14]
```

On peut définir un type « grands entiers naturels » par :

```
type big_nat = int list
```

Notons que ce type sera utiliser pour clarifier les signatures dans la suite du sujet, mais les grands entiers naturels ne sont que des `int list`! **On imposera les contraites suivantes sur les objets de type `big_nat` : tous les entiers dans la listes sont positifs et strictement inférieurs à m, le dernier entier étant, en plus, non nul**. Vous pouvez supposer que ces conditions sont toujours vérifiées pour les arguments des fonctions que vous écrivez (il n'est *pas* nécessaire de le vérifier). **Tous les résultats de vos fonctions doivent impérativement respecter ces mêmes conditions**, ce qui peut nécessiter beaucoup de précautions dans l'écriture de vos fonctions.

L'entier m a préalablement été défini dans OCaml, par une définition telle que

```
let m = 100
```

Vous pouvez vous servir de `m` dans n'importe laquelle de vos fonctions. On ne connaît pas la valeur exacte de `m` (mais on sait que c'est une puissance de dix supérieure ou égale à dix), aussi les fonction à écrire ne doivent pas faire d'hypothèse sur cette valeur (en particulier, on ne sait pas si $m = 100$!). La seule chose que l'on garantit est que l'on choisira m de sorte que $m^2 \leqslant$`max_int` (le produit de deux entiers inférieurs à m ne débordera jamais).

Pour représenter l'entier 0, **on utilisera une liste vide**. Cela nous permettra donc de représenter n'importe quel entier naturel.

5. Proposer une fonction `convert` de signature `int -> big_nat` (en d'autres termes, une fonction de signature `int -> int list`) prenant en argument un entier OCaml et renvoyant le « grand entier naturel » correspondant, soit une liste d'entiers le représentant dans le format précédemment décrit. Par exemple, `convert 1234567` doit renvoyer `[67; 45; 23; 1]` si `m`= 100, ou `[567; 234; 1]` si `m`= 1000, ou bien encore `[1234567]` si $m = 1000000000$.

6. Proposer une fonction `inc` de signature `big_nat -> big_nat` prenant en argument un grand entier naturel n et renvoyant le grand entier naturel $n + 1$. Par exemple, « `inc [34; 12]` » doit renvoyer « `[35; 12]` » (puisque $1234 + 1 = 1235$). On attend une complexité linéaire en la longueur de la liste passée en argument.

7. Proposer une fonction `add` de signature `big_nat -> big_nat -> big_nat` telle que « `add a b` » renvoie un grand entier naturel correspondant à la somme de a et b. Par exemple, « `add [34; 12] [87; 10; 5]` » devra renvoyer « `[21; 23; 5]` » puisque $1234 + 51087 = 52321$. On attend une complexité linéaire en la longueur des deux listes. On pourra s'inspirer de la façon de poser les additions apprise en primaire!

8. Proposer une fonction `dec` de signature `big_nat -> big_nat` prenant en argument un grand entier naturel n et retournant le grand entier naturel $n - 1$ si n est non nul, et déclenchera une erreur si n est nul (puisque de toute façon -1 n'est pas représentable). On attend une complexité linéaire en la longueur de la liste passée en argument.

9. Proposer une fonction `sub` de signature `big_nat -> big_nat -> big_nat` telle que « `sub a b` » renvoie un grand entier naturel correspondant à $a - b$ si $a \geqslant b$, et déclenchera une erreur sinon. Par exemple, « `add [21; 23; 5] [87; 10; 5]` » devra renvoyer

« `[34; 12]` » puisque 52321 − 51087 = 1234, tandis que « `add [87; 10; 5] [21; 23; 5]` » déclenchera une erreur. On attend une complexité linéaire en la longueur des deux listes.

10. Proposer une fonction `cmp` de signature `big_nat -> big_nat -> bool` telle que « `cmp a b` » retourne `true` si le grand entier naturel a est supérieur ou égal au grand entier naturel b, et `false` sinon.

2.3 Multiplications

Afin de pouvoir calculer une factorielle, on ne peut se contenter de sommes et de différences, il nous faudra des multiplications. Nous allons proposer ici un algorithme plus efficace que l'algorithme élémentaire vu en primaire, et utilisé par de nombreuses applications (dans une légère variante), par exemple par le langage Python pour effectuer ses multiplications entières.

Soit n et p deux entiers dont on souhaite connaître le produit. On suppose que l'on connaît un entier c tel que $n < m^{2c}$ et $p < m^{2c}$. On peut décomposer n et p en :

$$n = n_1 \times m^c + n_0 \qquad \text{et} \qquad p = p_1 \times m^c + p_0$$

où n_0, n_1, p_0 et p_1 sont des entiers dans $[\![0\,..\,m^c - 1]\!]$ (pour que cela ait un intérêt, il faut que n_1 ou p_1 au moins soit non nul).

Soit $q = n \times p$. On a naturellement

$$q = n_1 p_1 \times m^{2c} + \big(n_1 p_0 + p_1 n_0\big) \times m^c + n_0 p_0$$

On pose $q_2 = n_1 p_1$, $q_1 = n_1 p_0 + p_1 n_0$ et $q_0 = n_0 p_0$.

11. Donner trois encadrements précis des valeurs possibles pour q_0, q_1 et q_2.

12. Montrer que l'on peut calculer le coefficient q_1 à partir de q_0, q_2 et du produit $q_k = (n_1 + n_0)\big(p_1 + p_0\big)$, ce qui permet d'économiser une multiplication.

13. Considérons un grand entier naturel n non nul. Comment déterminer, à partir de sa représentation sous forme de liste d'entiers $\big[a_0; a_1; \dots a_{p-1}\big]$, le plus petit c tel que $n < m^{2c}$?

14. Soit un $c > 0$ tel que $n < m^{2c}$. Proposer une fonction `split` de signature `int -> big_nat -> big_nat * big_nat` prenant en argument c et n, et retournant le couple (n_0, n_1) tel que décrit précédemment.

15. Soit un $c > 0$, proposer une fonction `pow` de signature `int -> big_nat -> big_nat` prenant en argument c et un grand entier naturel q et retournant un grand entier naturel représentant $q \times m^c$.

Pour construire une fonction `mul` récursive permettant de multiplier deux grands entiers naturels n et p, on procède de la façon suivante :

- si $n = 0$ ou $p = 0$, le résultat est 0 ;
- si $n < m$ et $p < m$, on calcule directement $n \times p$ grâce à la multiplication entière OCaml (on sait que cela ne peut pas déborder) et on retourne le résultat sous forme de grand entier naturel ;
- sinon, on détermine le plus petit c vérifiant $n < m^{2c}$ et $p < m^{2c}$ puis
  - on détermine les grands entiers naturels n_0, n_1, p_0 et p_1 ;
  - on utilise la fonction `mul` (appel récursif) pour calculer q_0 et q_2 (un appel récursif pour chaque terme)
  - on utilise les fonctions `add` et `sub`, ainsi qu'un unique appel récursif à `mul` pour calculer q_1
  - on calcule $q = q_2 m^{2c} + q_1 m^c + q_0$ (sans appel à `mul`) et on retourne le résultat.

16. Proposer une implémentation de la fonction `mul`. On prendra soin de la rendre aussi lisible que possible, en faisant clairement apparaître les étapes précédemmen décrites.

L'algorithme de multiplication proposé ici est une variante d'un algorithme développé par Anatolii Alexevich Karatsuba en 1960 (l'algorithme original utilise le produit $q'_k = (n_1 - n_0)\big(p_1 - p_0\big)$ plutôt que q_k car il évite des problèmes de retenues ce qui le rend un peu plus efficace, mais nécessite de pouvoir travailler avec des valeurs négatives). Comme il n'effectue que trois appels récursifs, il permet de multiplier deux nombres de n chiffres avec une complexité que l'on peut établir en $O\big(n^{\log_2 3}\big)$, soit environ $O\big(n^{1.585}\big)$, ce qui est plus efficace que l'algorithme usuel appris en primaire qui nécessite $O\big(n^2\big)$ opérations.

2.4 Retour à notre problème

Rappelons que l'on dispose de fonctions `inc` et `dec` (`big_nat -> big_nat`) incrémentant et décrémentant un grand entier naturel, de fonctions `add`, `sub` et `mul` (`big_nat -> big_nat -> big_nat`) pour respectivement additionner, soustraire (sous réserve que le résultat soit positif) et multiplier deux grands entiers naturels, ainsi que d'une fonction `convert` (`int -> big_nat`) convertissant un entier OCaml en grand entier naturel.

17. Proposer une fonction `fact` de signature `int -> big_nat` qui prend en argument un entier OCaml n et renvoie un grand entier OCaml correspondant à sa factorielle ($n!$). Notons que l'argument est un entier OCaml car la factorielle d'un entier n plus grand que `max_int` demanderait de toute façon un temps considérable... en revanche, le résultat est un grand entier naturel, car $n!$ devient très vite très grand.

18. Écrire une fonction `count7_int` de signature `int -> int` prenant en argument un entier n et retournant le nombre de fois où le chiffre 7 apparaît dans son écriture décimale.

19. En déduire une fonction `count7_big_nat` de signature `int -> int` prenant en argument un grand entier naturel et retournant le nombre de fois où le chiffre 7 apparaît dans son écriture décimale.

Un appel à `count7_big_nat (fact 7)` donnera donc le résultat attendu, soit 560 apparitions du chiffre 7 dans 2024!. Il s'agit du troisième chiffre le plus fréquent dans l'écriture

décimale de 2024! après 0 et 6. C'est le chiffre 8 qui est le moins fréquent.

Si vous souhaitez vous en assurer par vous-même, voici ce que donne `List.iter (Printf.printf "%02d") (List.rev (fact 2024))` :

```
096460263446214926398052510226199790985387375890482144645958649662683760332267733851014700524912253548340662740320758479550870462950710541210
114256001203182619263078629977074321149302129154405032365449154188462564858837516772284331365982120543656814786353914637906013049028747757418
468322100672789667066615730603260622636356924703793570147797993547131319496078642460357268197320658715622239390607891890157067706779145414956
041673814738203466315647965131970045671578384728387034156940073715727760635796459029118365934356032352238846225237932063700491663741277148122
314527601584468737005048850304365143852284226398767699879623342462578604695003134417114511687579285325536111016330996051300430871823003300593
014037164050826913277309566027356198779759767471021235541114195745288538647903013893833047467824459181563503369617909095343580165374951232118
241306712347773406307741857691232291418296597635490315797583646305781206685022046282766171508495821464150090251539167261484286896838712372481
489179238671912921505425770425261838605845665267197913407862278426528965985262820287731333301834025228633658812081367908816946990403092581520
368143708743318721729876347356246608416500374753079055650483048841837841272676761906889614021097105058824428338117160941127604244793186458604
455458655855760284940836429035363285093069104770759612580379843158660083446728086708133990983975041486268208785796843904663516388627125424952
446270411395704547335095407764477860622896995399908272496176968841783028738879544837609111725719047908817535980555841792961912422299256008787
099560572736469484472140982528510846150405840394623051150713710139066160430076136827022902435938593216037851285916641113526569137590288035463
029188735503390387933570891442115703224772415188779452622347534612940438705275793091930118457661532044066216053077061323375063554502324441642
457713740797458137782640263894891437878719137517691674898243217907030052090651895959851365405298671480924162321147046796694677054397147498070
712453156890108384584358580265353992385521371145826722799820719004101463058033901430318180130801337514207158854542194919566082406442863543746
8666564240219600473073680007443547803173879998373098589876776779599572793561728263366307308373237771120357227016914987025978955136473245173531
097963984531930577847028441640813733886027143532735999162462242038297685895984915613932408092434650318700698380533398518906470316456602515334
125462072418898731409870680787148504689404265007175682907169143845077023595655250658192625492399236394999201807715962383951305372597882305079
585087638237380004739008416055980046042118667621966903020005762139041623609677497482146787787738293353337297066592878840807644462089512162439
136060854117196665411084506914787670057258645773079736657396686377931128544145435371891020871659295259982415054583878772459479843332097799936
729583810631337670102579822411981740356555198768792466787602627645578114690649614212348075407594868695855763666912895052482541865676073732669
042067494726792747850583323116032353918882002490065044155978237129773743603155820199223204882116656736747111422652649209273980801098703605759
204622910794369264653030236748756291763620302590371784026985332599997367433875265659080040077152601759240871276233406871274350853410219568093
767375976127953629845616455683239280615130379286000760813973852900564593812266820187863395431097371783494563671201953944142330268858140344898
735585756278188877839171671493534189114189679449554976777562547563108504968274014247805902294529075583416555296085730470356715305956760172527
415316186764650735860044943360657786375599873289325712969497717130929173036731039449145274815347613243666977166204109166005976179310579009128
539821915882710350325776845339552716966255844588879664165153164153827514210901809995515559486766671101078101531513311801373936292021813735381
196243942297450745397435147879414647193078511495554993310622021078013091740934850152455573649186325957660632319869252402791470598653390306887
894852248070332628105353063922183216232919901134163999402264720594000857685009006040029523861996386611235492875281695009666270112797583066665
057895327219529917476129777240719323502207653948944606234913430730500582162842752888985544043284420294140479709096516387527912091914494425594
159523459087717092120743627374610950326654324252695481098366716196320802715867689629315830690741076887639684759622975986646949655510885584694
109829985947537285233037538613498658508631226256762725088159265473395847060711299199162105091802668504461157646761834566119728197110872011264
553459513684188962639697997320808885412134945451749587992912262128293557070253362395549676231971874917943424068104940911070940868082868088034
096112180442971490795781014612598692147147243075823336234834088234839597756376235869995100458920914695331966917746666038260298092121095220423
679022285922587343720244626345153811302436933869428379545911580507809347967495313228726046882885223181937169874192323779368275275109949715105
777563253843729918009577959954109635038804929138520764601411956695355529840492279367480699758280223801472150926013056644956096265827149090487
666820574934221633725217216349569648398577491166685908593117644512856291566656982001106010901794771100669396016835980126069291747128095059661
75075264288286697659251819221640168491972915576610350929657226905088676218555479820245777912954880000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000
```