Passeport pour l'informatique

EXERCICES D'ALGORITHMIQUE

Oraux d'ENS

Ouvrage collectif — Coordination Jean-Claude Bajard

INTERNATIONAL™ THOMSON PUBLISHING

Jean-Claude Bajard, Hubert Comon, Claire Kenion,
Daniel Krob, Michel Morvan, Jean-Michel Muller,
Antoine Petit, Mattéo Rizza-Murgier et Yves Robert

Coordination : Jean-Claude Bajard

Exercices d'algorithmique

Oraux d'ENS

INTERNATIONAL THOMSON PUBLISHING FRANCE
An International Thomson Publishing Company

E-mail : *contact@itp.fr*
Listes de diffusion : *listserver@itp.fr*
World-Wide Web : *http ://www.itp.fr*

Paris • Albany • Belmont • Bonn • Boston • Cincinnati • Detroit • Johannesburg
Londres • Madrid • Melbourne • Mexico • New York • Singapour • Toronto • Tokyo

1

Vers la récursivité

« *Diviser pour régner. Faire moins, pour faire plus.* »

1.1 L'angoisse de la panne sèche

Une route comporte n stations-service. La première est à une distance d_1 du départ, la la deuxième est à une distance d_2 de la première, la troisième à une distance d_3 de la deuxième, etc. La fin de la route est à une distance d_{n+1} de la n-ième station-service. Les distances d_i sont représentées par un tableau de flottants.

Un automobiliste prend le départ de la route avec une voiture dont le réservoir d'essence est plein. Sa voiture est capable de parcourir une distance r (mais pas plus !) avec un plein. On suppose que r est supérieur ou égal à chacun des d_i et inférieur à leur somme, sinon le problème n'a pas de sens.

Question 1

L'automobiliste désire faire le plein le moins souvent possible. Écrire une fonction OCaml qui détermine à quelles stations-service il doit s'arrêter.

Question 2

Maintenant, l'automobiliste part avec un réservoir vide. Il doit au départ acheter de l'essence (autant qu'il veut dans la contenance de son réservoir) au prix de E_0 euros par litre. Par la suite, à la station numéro i, l'essence coûte E_i euros par litre. L'automobiliste consomme kt litres pour parcourir une distance t, et le réservoir peut contenir L litres d'essence. Écrire une fonction OCaml qui indique à quelles stations-service l'automobiliste doit s'arrêter, et combien de litres il doit prendre à chaque fois, pour que son trajet lui coûte le moins cher possible.

CORRIGÉ

Question 1

Un algorithme glouton est optimal : on regarde jusqu'où on peut aller et on s'arrête à la première station avant ce point, puis on recommence.

```
let rapide (d:int array) (r:int) : int list =
  let n = Array.length d in
  let rec aux acc i reste =
    if i=n then acc else
    if d.(i) > reste then
      aux (i::acc) (i+1) (r-d.(i))
    else
      aux acc (i+1) (reste-d.(i))
  in List.rev (aux [] 0 r)
```

Cet algorithme est bien optimal. En effet, soit une autre solution. Son premier plein est forcément au même endroit ou avant le premier plein de la solution de l'algorithme. Par récurrence, son i-ième plein se produit au même endroit ou avant le i-ième plein de l'algorithme.

Question 2

Repérons tout d'abord la station où l'essence est la moins chère, et faisons deux remarques :

- L'automobiliste doit faire en sorte que son réservoir soit vide au moment où il arrive à cette station (risqué en pratique !). En effet, s'il lui reste x litres d'essence à ce moment là, il les a payés plus cher auparavant pour rien.
- Toujours à cette station, il doit prendre juste ce qu'il faut d'essence pour aller jusqu'au bout si la capacité de son réservoir le permet, soit remplir complètement son réservoir.

On recommence pour les deux sous-problèmes consistant à aller du début à cette station-service et de cette station service à la fin. Pas besoin de récursivité, la stratégie qui s'en déduit est la suivante. À chaque fois que l'on passe devant une station service :

- Soit il existe dans le rayon d'action que peut atteindre la voiture avec un plein des stations moins chères, dans ce cas, on prend le minimum d'essence (éventuellement pas du tout) permettant d'aller à la première station moins chère que celle où l'on se trouve (attention : la première, c'est-à-dire pas nécessairement la moins chère des stations accessibles).
- Soit toutes les stations que l'on peut atteindre sont plus chères, dans ce cas on remplit complètement le réservoir, sauf si on peut atteindre l'arrivée, dans quel cas on prend juste ce qu'il faut pour y aller.

L'écriture de la fonction OCaml ne représente plus aucune difficulté, elle est laissée au lecteur.

⋆⋆⋆⋆⋆⋆⋆⋆

1.2 Produit de plusieurs matrices

On cherche à effectuer un produit de matrices

$$M_1 \times M_2 \times ... \times M_n$$

où la matrice i comporte r_{i-1} lignes et r_i colonnes, en effectuant la moins possible de multiplications de réels (la multiplication de deux matrices se fera par la méthode usuelle).

Question 1

Écrire une fonction qui multiplie une matrice A de taille $p \times q$ par une matrice B de taille $q \times r$. Combien de multiplications de réels nécessite-t-elle ?

Question 2

On effectue le produit $M_1 \times M_2 \times M_3 \times M_4$, où M_1 est de taille 10×20, M_2 de taille 20×50, M_3 de taille 50×1 et M_4 de taille 1×100. Combien de multiplications de réels fera-t-on si on calcule le produit dans les deux ordres (indiqués par les parenthèses) suivants :

$$M_1 \times (M_2 \times (M_3 \times M_4))$$

$$(M_1 \times (M_2 \times M_3)) \times M_4$$

Question 3

Écrire une fonction pour trouver le plus petit nombre de multiplications de réels nécessaires pour calculer $M_0 \times M_1 \times ... \times M_{n-1}$, où la matrice i comporte r_{i-1} lignes et r_i colonnes. Aide : définir une valeur `m.(i).(k)` égale au plus petit nombre de multiplications pour calculer $M_i \times M_{i+1} \times ... \times M_k$.

CORRIGÉ

Question 1

On a classiquement :

```
type matrix = float array array
let produit (a:matrix) (b:matrix) : matrix =
  let p = Array.length a in
  let q = Array.length b in
  let r = Array.length b.(0) in
  let c = Array.make_matrix p r 0. in
  for i=0 to p-1 do
    for j=0 to r-1 do
      for k=0 to q-1 do
        c.(i).(j) <- c.(i).(j) +. a.(i).(k) *. b.(k).(j)
```

```
      done
    done
  done;
  c
```

Cette fonction utilise $p \times q \times r$ multiplications.

Question 2

Il faut $50 \times 1 \times 100 + 20 \times 50 \times 100 + 10 \times 20 \times 100 = 125\,000$ multiplications pour la première méthode et $20 \times 50 \times 1 + 10 \times 20 \times 1 + 10 \times 1 \times 100 = 2200$ pour la seconde méthode.

Question 3

Il suffit de voir comment s'obtient `m.(i).(j)`. On trouve aisément :

$$\texttt{m.(i).(j)} = \begin{cases} 0 & \text{si } i = j \\ \min_{i \leq k < j} \{\texttt{m.(i).(k)} + \texttt{m.(k+1).(j)} + r_{i-1} r_k r_j\} & \text{sinon} \end{cases}$$

Il n'est pas nécessaire d'écrire une fonction récursive, on itère sur la longueur des chaînes de multiplication.

```
let nb_min_multiplications (r:int array) : int =
  let n = Array.length r - 1 in
  let m = Array.make_matrix n n 0 in

  for l = 2 to n do
    for i = 0 to n - l do
      let j = i + l - 1 in
      m.(i).(j) <- max_int;
      for k = i to j - 1 do
        let q = m.(i).(k) + m.(k+1).(j) + r.(i) * r.(k+1) * r.(j+1) in
        if q < m.(i).(j) then
          m.(i).(j) <- q
      done
    done
  done;
  m.(0).(n-1)
```

⋆⋆⋆⋆⋆⋆⋆⋆

1.3 Les deux points les plus proches

Le but de ce problème est d'identifier, dans un ensemble de points, quel est le couple de points les plus proches au sens de la distance euclidienne. Ce type d'algorithme voit son utilité dans les transports aériens ou maritimes.

L'ensemble de points est vu comme un tableau de 2-tuples de coordonnées (x, y).

On dispose de la fonction `sqrt (x:float) : float` qui renvoie la racine carrée de `x`.

Question 1

Donner un algorithme itératif qui détermine le couple de points les plus proches de l'ensemble.

Montrer qu'à chaque itération, votre algorithme répond aux conditions que vous vous êtes fixées (c'est-à-dire vérifie un invariant de boucle).

Évaluer le nombre d'additions, de multiplications et d'appels à la fonction `sqrt`.

L'ensemble de points est stocké dans deux tableaux `x` où les points sont ordonnés suivant les abscisses croissantes, et `y` où les points sont ordonnés suivant les ordonnées croissantes.

Question 2

Écrire une fonction OCaml `decoupe (x:point array) (y:point array) (y':point array) (deb:int) (fin:int) : unit` qui place les points du sous-tableau `x`[`deb`..`med`] classés par ordonnée croissante dans le sous-tableau `y'`[`deb`..`med`] et les éléments du sous-tableau `x`[`med+1`..`fin`] classés par ordonnée croissante dans le sous-tableau `y'`[`med+1`..`fin`], où `med` $= \left\lfloor \frac{\texttt{deb}+\texttt{fin}}{2} \right\rfloor$.

Évaluer le nombre de transferts effectués lors de l'exécution de cette fonction.

Question 3

Nous supposons que A et B sont les points les plus proches du sous-tableau `x`[`deb`..`med`] et, C et D sont les points les plus proches du sous-tableau `x`[`med+1`..`fin`].

Écrire un algorithme linéaire qui détermine les deux points les plus proches de l'ensemble `x`[`deb`..`fin`], connaissant A, B, C, D et `y`[`deb`..`fin`].

En déduire une fonction de recherche du couple de points les plus proches (où A, B, C et D sont déterminés de façon classique).

Déterminer la complexité de votre fonction.

Que peut-on conclure ?

CORRIGÉ

Question 1

Nous proposons un algorithme exhaustif où tous les couples de points sont examinés.

```
type point = int * int

let dist_sq (p1:point) (p2:point) : int =
  let x1, y1 = p1 and x2, y2 = p2 in
  (x2-x1)*(x2-x1) + (y2-y1)*(y2-y1)
```

```
let couple (tab:point array) : (point * point) =
  let n = Array.length tab in
  let min_dist = ref max_int in
  let p1 = ref (0,0) and p2 = ref (0,0) in

  for i=0 to n-2 do
    for j=i+1 to n-1 do
      let dist = dist_sq tab.(i) tab.(j) in
      if dist < !min_dist then (
        min_dist := dist;
        p1 := tab.(i);
        p2 := tab.(j)
      )
    done
  done;
  (!p1, !p2)
```

Dans la boucle i : `p1`, `p2` est le couple de points les plus proches avec `p1` pris parmi les i premiers points de `tab`.
Dans la boucle j : `p1`, `p2` est le couple de points les plus proches avec `p1` pris parmi les $i-1$ premiers points de `tab` avec `p1=tab.(i)` et `p2` pris parmi les j premiers points de `tab`.

Le nombre d'additions est $3 \times (n-1)n/2$, de multiplications $(n-1)n$, où n est le nombre de points. Il n'y a pas besoin d'appeler la fonction `sqrt`.

Question 2

Il s'agit simplement de transcrire ce qui est proposé dans l'énoncé :

```
let decoupe (x:point array) (y:point array) (y':point array)
    (deb:int) (fin:int) : unit =
  let med = (deb + fin) / 2 in
  let i = ref deb in
  let j = ref (med + 1) in
  for k=deb to fin do
    if fst y.(k) > fst x.(med) then (
      y'.(!j) <- y.(k);
      incr j
    )
    else (
      y'.(!i) <- y.(k);
      incr i
    )
  done
```

Le nombre de transferts est de l'ordre du nombre d'éléments de `x`.

Question 3

Suivant les instructions de l'énoncé, le tableau `x` est découpé en deux parties. Dans

chacune d'entre elles, nous avons recherché le couple de points les plus proches.

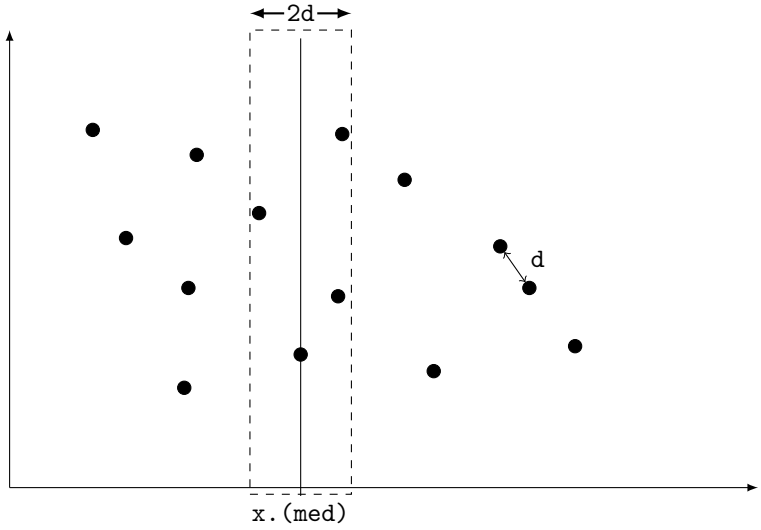
Mais la distance la plus courte peut être donnée par un couple de points à cheval sur les deux parties.

Supposons que nous ayons comparé les deux distances et gardé le couple donnant la plus petite distance `dist`, alors via la fonction `termine`, nous allons rechercher s'il existe un couple de points pris dans chacune des deux parties, dont la distance est plus petite.

Nous reprenons la même terminologie pour les variables.

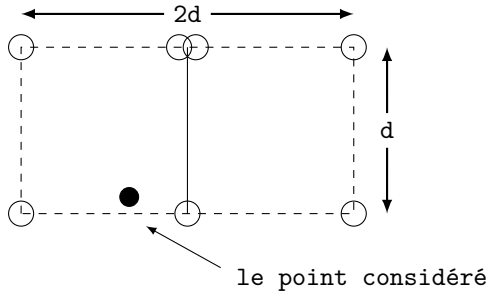
```
let termine (x:point array) (y:point array) (deb:int) (fin:int)
    (p1:point) (p2:point) : (point * point) * int =
  let dist = ref (dist_sq p1 p2) in
  let n = Array.length x in
  let z = Array.make n (0,0) in
  let t = ref 0 in

  let med = (deb + fin) / 2 in
  let a = fst x.(med) - !dist in
  let b = fst x.(med) + !dist in
  (* On ajoute tous les candidats potentiels *)
  for k=deb to fin do
    if a < fst y.(k) && fst y.(k) < b then (
      z.(!t) <- y.(k);
      incr t
    )
  done;
  (* On cherche une plus petite distance *)
  let p1 = ref p1 and p2 = ref p2 in
  for k=0 to !t-2 do
    let i = ref 1 in
    while !i + k < !t && (snd z.(k + !i)) - (snd z.(k)) < !dist do
      let d = dist_sq z.(k) z.(k + !i) in
      if d < !dist then (
        p1 := z.(k);
        p2 := z.(k+1);
        dist := d
      );
      incr i
    done
  done;
  (!p1, !p2), !dist
```

On ne considère que les points contenus dans le rectangle en pointillés, en partant du point le plus bas.



Si `d` est la plus petite distance pour chacune des deux moitiés, nous avons au plus 4 points dans un carré de côté `d`.

Ainsi, dans la boucle `while`, il y a au plus 7 itérations. On représente ici le rectangle examiné dans la boucle `while` :



Nous pouvons ainsi envisager la fonction récursive suivante :

```
let cherche (x:point array) (y:point array) : point * point =
  let n = Array.length x in
  let y' = Array.copy y in

  let rec cherche_aux deb fin =
    if fin = deb then
      ((0,0), (0,0)), max_int
    else if fin = deb + 1 then
      (x.(deb), x.(fin)), dist_sq x.(deb) x.(fin)
    else (    (* fin > deb+1 *)
      let med = (deb+fin)/2 in
```

```
      decoupe x y y' deb fin;
      let (a, b), d1 = cherche_aux deb med in
      let (c, d), d2 = cherche_aux (med+1) fin in
      let p1, p2 = if d1 < d2 then  a, b else c, d in
      termine x y deb fin p1 p2
    )
  in let (p1, p2), d = cherche_aux 0 (n-1) in
  p1, p2
```

Les fonctions `decoupe` et `termine` ont une complexité linéaire (c'est à dire en $O(n)$). La complexité de la fonction `cherche` lorsque `deb=0` et `fin=n-1` est donnée par la formule de récurrence suivante : $C(n) = 2 \cdot C(n/2) + O(n)$.

La complexité de `cherche` est donc en $O(n \log_2 n)$.

⋆⋆⋆⋆⋆⋆⋆⋆

1.4 Hanoï

Les tours de Hanoï est un exemple très connu pour sa programmation récursive. Nous nous intéressons ici à sa version itérative.

Le jeu en lui même est très simple. Nous disposons de trois colonnes sur lesquelles sont empilés des disques de taille différentes.

Les règles sont les suivantes : sur chaque colonne, un disque plus gros ne peut pas être posé sur un disque plus petit, on ne peut déplacer qu'un seul disque à la fois.

Au départ, tous les disques sont empilés sur la première colonne. Le joueur doit en un minimum de coups déplacer tous les disques de la première colonne à la dernière colonne.

Question 1

Donner les déplacements à faire avec trois disques. Puis avec quatre disques.

Question 2

Montrer que le plus petit disque se déplace de la même façon un coup sur deux. Montrer que pour chaque autre déplacement, un seul choix est possible.

Question 3

Numérotons les coups en partant de 1.

Déterminer la relation qu'il y a entre le numéro du disque à déplacer et le numéro du coup.

Pour n disques, combien de déplacements effectuez-vous ? Est-ce optimal ?

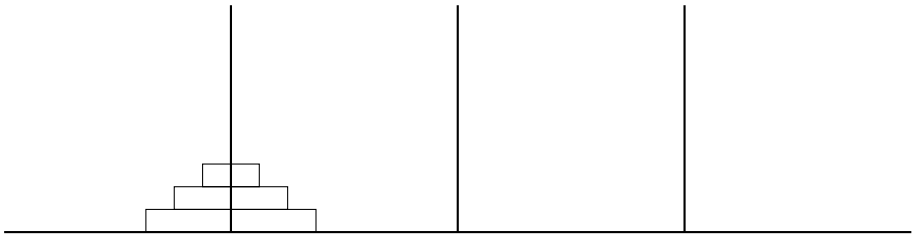
Écrire le programme affichant tous les coups effectués.

CORRIGÉ

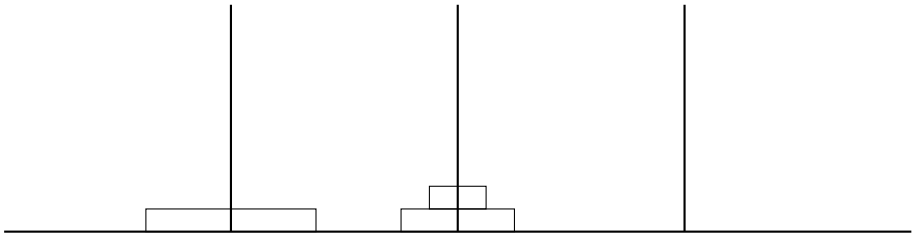
Question 1

Nous donnons dans les tableaux suivants les positions de chaque disque au cours du temps. Les colonnes sont numérotées 0, 1 et 2. Les disques sont nommés du plus petit au plus grand d_0, d_1, ..., d_n. Tous les disques sont sur la colonne 0 au départ.

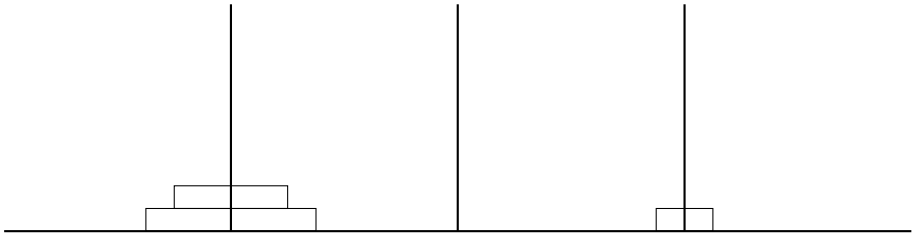
Pour trois disques nous avons :



Comme nous devons déplacer le plus grand disque de la colonne 0 à la colonne 2, la configuration suivante est un passage obligé :



Pour obtenir cette configuration, le disque immédiatement inférieur au plus grand a dû être déplacé de la colonne 0 sur la colonne 1 d'où un autre passage obligé :



Nous en déduisons le tableau suivant :

coups	0	1	2	3	4	5	6	7
d_0	0	2	2	1	1	0	0	2
d_1	0	0	1	1	1	1	2	2
d_2	0	0	0	0	2	2	2	2

Pour quatre disques, nous avons, en suivant le même raisonnement :

coups	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
d_0	0	1	1	2	2	0	0	1	1	2	2	0	0	1	1	2
d_1	0	0	2	2	2	2	1	1	1	1	0	0	0	0	2	2
d_2	0	0	0	0	1	1	1	1	1	1	1	1	2	2	2	2
d_3	0	0	0	0	0	0	0	0	2	2	2	2	2	2	2	2

Question 2

Les deux exemples précédents mettent en évidence une dépendance entre la parité du nombre de disques et le sens de déplacement de ceux-ci.

Tous les coups impairs, le disque d_0 se déplace de la colonne x vers la colonne $x+(-1)^n$ mod 3, où n représente le nombre de disques.

Comme on ne peut rien mettre sur le disque d_0, le seul choix possible est de déplacer le plus petit des disques situés sur les deux autres colonnes. Maintenant, seul le disque d_0 est déplaçable sinon il y a retour à la situation précédente. Nous vérifions ainsi que d_0 est à déplacer un coup sur deux.

Question 3

En fait, le disque d_1 se déplace comme s'il était le disque 0 des $n-1$ disques restants en ne prenant en compte que les coups pairs. Ce qui revient à un déplacement tous les coups pairs (multiple de deux) non multiples de quatre.

Ainsi, le disque d_i se déplace de $(-1)^{n-1}$ tous les coups multiples de 2^i non multiples de 2^{i+1}.

Ainsi, le disque d_{n-1} se déplace au coup 2^{n-1} lorsque tous les autres disques sont sur la colonne 1. Le temps de les replacer sur la colonne 2 est le même que celui pour les placer sur 1, c'est-à-dire $2^{n-1}-1$.

D'où le temps total pour n disques : 2^n-1. On peut maintenant le montrer par récurrence.

Est-il optimal ? Oui, car pour déplacer le plus grand des disques, il faut avoir déplacé tous les autres sur l'autre colonne, ce qui donne la formule de récurrence suivante, où $T(n)$ représente le nombre de coups pour n disques :

$$T(n) = 2 \cdot T(n-1) + 1$$

Nous obtenons $T(n) = 2^n - 1$.

Voici un exemple de programme où :

```
let deplacements (n:int) : unit =
  let p = Int.shift_left 1 n in (* 2^n *)
  for i=1 to p-1 do
    let t = ref i and k = ref 0 in
    while !t mod 2 = 0 do
      t := !t / 2;
      incr k
    done;
    let num_disque = !k in
    let deplacement = -2*((n - !k) mod 2) + 1 in
    (* Coup à jouer *)
    let dir = if deplacement = -1 then "gauche" else "droite" in
    Printf.printf "Déplacer le disque %d vers la %s\n" num_disque dir
  done
```

La complexité en termes déplacement est optimale. On peut calculer le nombre de comparaisons de la façon suivante (les seules comparaisons effectuées sont dans la condition de la boucle `while`) :

- La moitié des nombres binaires se terminent par un 1. Dans ce cas, on effectue une comparaison.
- Parmi ceux se terminant par un 0, la moitié possède un 1 en deuxième position. Dans ce cas, on effectue 2 comparaisons.
- Parmi ceux se terminant par deux zéros, la moitié possède un 1 en troisième position. Dans ce cas, on effectue 3 comparaisons.
- De manière récurrente, on a donc une proportion $\frac{1}{2^n}$ des nombres qui nécessitent n itérations.

Donc en sommant sur toutes les longueurs, le nombre d'itérations N est donné par la formule :

$$N = 2^n \cdot \sum_{k=1}^{n} k \frac{1}{2^k} = 2^{n-1} \cdot \sum_{k=1}^{n} k \left(\frac{1}{2}\right)^{k-1}$$

En remarquant que :

$$\sum_{k=1}^{n} k x^{k-1} = \frac{\mathrm{d}}{\mathrm{d}x} \sum_{k=1}^{n} x^k = \frac{\mathrm{d}}{\mathrm{d}x} \frac{x - x^{n+1}}{1 - x}$$

on obtient $N = 2^{n+1} - (n + 2)$.

On peut si on le désire également donner la version récursive suivante où A, B et C représentent les trois colonnes :

`Hanoï`(n, A, B, C)
    si $n \geq 1$ faire
        `Hanoï`($n - 1$, A, C, B)
        Déplacer le disque de A vers C
        `Hanoï`($n - 1$, B, A, C)

⋆⋆⋆⋆⋆⋆⋆⋆