Semaine 2 : Série d'exercices sur les algorithmes [Solutions]

1 Quel est le bon algorithme ?

Le bon algorithme est le **1.B**. Le **1.A** ne calcule que la somme des $n/2$ premiers nombres pairs ; le **1.C** calcule la somme des $2n$ premiers nombres entiers, et le **1.D** est incorrect pour deux raisons : tout d'abord la valeur de s n'est pas initialisée (le résultat est donc indéfini) ; mais même si l'on suppose la convention que toute variable est initialisée à 0 [1], le résultat serait aussi incorrect car correspondrait à la *moitié* de la somme des n premier entiers pairs.

2 Que font ces algorithmes ?

a) La sortie de l'algorithme **2.A** vaut 2^n (on répète n fois l'opération $i \longleftarrow 2i$ en partant de $i = 1$), et le nombre d'opérations effectuées est donc de l'ordre de n.

b) Quelles que soient les valeurs de a et b, la sortie de l'algorithme **2.B** vaut $a \cdot b$. Le nombre d'opérations effectuées est de l'ordre du minimum de a et b. En effet, si $a < b$, alors on répète a fois l'opération $s \longleftarrow s + b$; dans le cas contraire, on répète b fois l'opération $s \longleftarrow s + a$. Vu qu'on part de $s = 0$, le résultat est le même dans les deux cas, et le nombre d'additions effectuées est toujours le mininum.

3 Au temps des Egyptiens

3.1 Comprendre

Exemple de déroulement pour $a = 5$ et $b = 7$:

x	y	z
5	7	0
5	6	5
10	3	5
10	2	15
20	1	15
20	0	35

Cet algorithme permet de calculer le produit entre deux nombres entiers. Pour $a, b \geq 1$: $\mathbf{algo}(a, b) = a \times b$.

3.2 Démontrer

Il est possible de le démontrer formellement par récurrence. Une version de cette technique permet de démontrer qu'une proposition $P(n)$ est vérifiée pour tout nombre entier n en procédant comme suit :

1. on démontre que $P(n)$ est vérifiée pour un certain entier n connu ; par exemple $n = 1$ ou $n = 2$; on démontre cela par exemple par calcul direct ;

1. Ce que nous **ne** ferons **pas** dans ce cours (ni dans ses examens).

2. ensuite, on démontre que, si $P(i)$ est vérifiée pour tout entier inférieur ou égal à n, alors $P(n+1)$ est aussi vérifiée.

Dans notre cas, avant d'entrer dans la récurrence, montrons tout d'abord que, pour tout $a \in \mathbb{N}^*$, pour tout constante $k \in \mathbb{N}^*$ et pour tout $b \in \mathbb{N}^*$ on a :

$$\mathbf{algo}(k \times a, b) = k \times \mathbf{algo}(a, b). \tag{1}$$

En effet, les opérations impliquant (directement ou indirectement) la valeur de la première entrée (a) sont :

— l'initialisation $x \longleftarrow a$; si l'on remplace a par $k \times a$ alors x sera aussi k fois plus grand ;
— la multiplication par 2 de x si y est pair : là aussi, si x est k fois plus grand au départ, il l'est encore par la suite ;
— l'addition de x à z. Comme 0 (valeur initiale de z) est égal à $k \times 0$, cette opération ($z \longleftarrow z + x$), qui est la seule sur z, fera également que z sera k fois plus grand si x est k fois plus grand ; or z est justement le résultat de l'algorithme.

On peut alors commencer la preuve par récurrence sur b pour a fixé :

1. On vérifie par calcul direct que, pour $b = 1$ ou même $b = 2$, on a effectivement : $\mathbf{algo}(a, b) = a \times b$.
2. Soit $b \in \mathbb{N}^*$. On cherche à démontrer que, si $\forall a \in \mathbb{N}^*\ \forall i \in \mathbb{N}^*\ i \leq b\ \mathbf{algo}(a, b) = a \times b$, alors $\mathbf{algo}(a, b+1) = a \times (b+1)$. Pour cela, on peut procéder comme suit :
   (a) si $b+1$ est pair, i.e. il existe c tel que $b + 1 = 2\,c$, alors :

   $$\mathbf{algo}(a, b+1) = \mathbf{algo}(a, 2\,c) = 2\,a \times c, \tag{2}$$

   En effet, par le premier branchement conditionnel de l'algorithme on a $\mathbf{algo}(a, 2\,c) = \mathbf{algo}(2\,a, c)$ (puisque $2\,c$ est un nombre pair). Or $\mathbf{algo}(2\,a, c) = 2 \times \mathbf{algo}(a, c)$ par le résultat (1) ci-dessus. Et comme $c \leq b$, on a alors par hypothèse de récurrence $\mathbf{algo}(a, c) = a \times c$.
   Donc si $b+1$ est pair, on a bien $\mathbf{algo}(a, b+1) = a \times (b+1)$.
   (b) Si $b+1$ est impair, i.e. il existe c tel que $b + 1 = 2\,c + 1$, alors :

   $$\mathbf{algo}(a, b+1) = \mathbf{algo}(a, 2\,c+1) = a \times (2\,c+1) = a \times (b+1)$$

   En effet, puisque $2\,c + 1$ est un nombre impair, par la 2$^{\text{e}}$ partie du branchement conditionel de l'algorithme, nous avons

   $$\mathbf{algo}(a, 2\,c+1) = a + \mathbf{algo}(a, 2\,c)$$

   Ensuite, en utilisant le résultat (2), on obtient bien que $\mathbf{algo}(a, 2\,c+1) = a + 2\,a \times c = a \times (2\,c+1)$.

Ceci conclut la démonstration pour tous les cas. On a donc bien :

$$\forall a \in \mathbb{N}^*\ \forall b \in \mathbb{N}^*\ \mathbf{algo}(a, b) = a \times b$$

4 Et que fait celui-ci ?

a) La sortie est donc $|22 - (-4)| = 26$.

b) Cet l'algorithme calcule la plus grande différence en valeur absolue entre tous les nombres de la liste. En effet, les deux boucles de l'algorithme parcourent toutes les paires de nombres de la liste, et la valeur de x, initialisée à $|L(2) - L(1)|$, est augmentée à chaque fois que la différence (en valeur absolue) entre les deux nombres considérés est plus grande que la valeur courante de x.

c) Vu qu'il y a deux boucles imbriquées qui parcourent chacune toute la liste de taille n, le nombre d'opérations est de l'ordre de n^2. Plus précisément, on peut voir que l'instruction « ***Si*** » est exécutée $\frac{n(n-1)}{2}$ fois, ce qui est bien de l'ordre de n^2.

d) Oui, on peut faire mieux ici. Pour cela, il est important de réaliser qu'il n'y a pas besoin de comparer toutes les paires de nombres de la liste pour trouver la plus grande différence. On peut améliorer l'algorithme de deux façons différentes :

1. Première solution : l'algorithme parcourt une seule fois la liste et garde au fur et à mesure la trace de la plus petite (disons $x_{\min}$) et de la plus grande (disons $x_{\max}$) des valeurs rencontrées. A la fin de l'algorithme, la plus grande différence n'est rien d'autre que $x_{\max} - x_{\min}$. Cet algorithme n'utilise que de l'ordre de n opérations.

algorithme 4 amélioré, version 1

entrée : *L liste de nombres entiers, n taille de la liste*
sortie : *la plus grande différence (en valeur absolue)*

```
x_min ⟵ L(1)
x_max ⟵ L(1)
Pour i allant de 2 à n
    Si L(i) > x_max
        x_max ⟵ L(i)
    Si L(i) < x_min
        x_min ⟵ L(i)
Sortir : x_max − x_min
```

2. Deuxième solution : trier au préalable la liste dans l'ordre croissant avec un algorithme de tri efficace qui n'a besoin que de l'ordre de $n \log n$ opérations. La plus grande différence est alors égale au dernier nombre de la liste triée moins le premier nombre de cette même liste. Cet algorithme utilise donc en tout de l'ordre de $n \log(n)$ opérations.

algorithme 4 amélioré, version 2

entrée : *L liste de nombres entiers, n taille de la liste*
sortie : *la plus grande différence (en valeur absolue)*

```
L' ⟵ tri efficace(L)
Sortir : L'(n) − L'(1)
```

5 Réparez-moi ces algorithmes !

a) Le problème de l'algorithme de Jeanne est le suivant : tout se passe bien jusqu'à ce que $i = 1$ et $j = 5$, moment auquel l'algorithme rajoute 5 à s et recommence la boucle, en passant à $i = 2$. A ce 2^e passage, comme la valeur de j *n'a pas changé* (elle est toujours égale à 5), l'algorithme n'entre pas dans la boucle « ***Tant que*** » et ajoute de nouveau 5 à s, et ainsi de suite. La sortie de l'algorithme sera donc $s = 5n$ et non le résultat escompté. Pour remédier à ce problème, il faut incrémenter j après avoir mis à jour la valeur de s, comme suit :

algorithme de Jeanne corrigé

entrée : n *nombre entier positif*
sortie : s

$s \longleftarrow 0$
$j \longleftarrow 1$
Pour i *allant de* 1 *à* n
  Tant que j *n'est ni un multiple de* 5 *ni un multiple de* 7
    $j \longleftarrow j + 1$
  $s \longleftarrow s + j$
  $j \longleftarrow j + 1$
Sortir : s

Ainsi, après avoir pris la valeur $j = 5$, j passe à 6, qui n'est plus ni un multiple de 5 ni de 7, et l'algorithme rentre dans la prochaine boucle "Tant que".

b) Le problème de l'algorithme de Jean est le suivant : il existe des nombres qui sont à la fois des mutiples de 5 et de 7 (35, par exemple). De tels nombres seront comptabiliés deux fois dans la somme s par l'algorithme. On peut réparer ce problème de deux manières :

algorithme de Jean corrigé, version 1

entrée : n *nombre entier positif*
sortie : s

$s \longleftarrow 0$
$i \longleftarrow 0$
$j \longleftarrow 0$
Tant que $i < n$
  $j \longleftarrow j + 1$
  Si j *est un multiple de* 5
    $s \longleftarrow s + j$
    $i \longleftarrow i + 1$
  Sinon
    Si j *est un multiple de* 7
      $s \longleftarrow s + j$
      $i \longleftarrow i + 1$
Sortir : s

algorithme de Jean corrigé, version 2

entrée : n *nombre entier positif*
sortie : s

$s \longleftarrow 0$
$i \longleftarrow 0$
$j \longleftarrow 0$
Tant que $i < n$
  $j \longleftarrow j + 1$
  Si j *est un multiple de* 5 *ou de* 7
    $s \longleftarrow s + j$
    $i \longleftarrow i + 1$
Sortir : s

6 Création d'algorithmes

6.1 La plus petite valeur

Il suffit de regarder tour à tour toutes les valeurs, en mémorisant la plus petite vue jusqu'ici :

La plus petite valeur

entrée : *Liste L d'entiers* ***non vide***
sortie : *a, plus petite valeur dans L*

```
a ⟵ L(1) (premier élément de L)
t ⟵ taille(L)
Pour i de 2 à t
    Si a > L(i)
        a ⟵ L(i)
sortir : a
```

Cet algorithme est en $\mathcal{O}(n)$, où n est la taille de la liste. Il fonctionne aussi bien avec des valeurs *ex aequo*.

6.2 La plus petite différence

Une approche possible consiste à tester les différences entre toutes les paires possibles :

La plus petite différence-1

entrée : *Liste L d'entiers contenant au moins 2 éléments*
sortie : *a, b nombres avec la plus petite différence*

```
δ_min ⟵ ∞
t ⟵ taille(L)
Pour i de 1 à t − 1
    Pour j de i + 1 à t
        δ ⟵ |L(i) − L(j)| (valeur absolue)
        Si δ < δ_min
            δ_min ⟵ δ
            a ⟵ L(i)
            b ⟵ L(j)
sortir : (a, b)
```

Calculons la complexité temporelle de cet algorithme. Soit n le nombre d'éléments dans la liste.

- Toutes les assignations prennent un temps constant, donc leur complexité temporelle est $\mathcal{O}(1)$.
- Calculer la taille d'une liste est $\mathcal{O}(n)$.
  Note : certains diraient « c'est au pire $\mathcal{O}(n)$ », mais cette notion de « au pire » est déjà inclue dans la notation $\mathcal{O}$, par exemple 1 (constante) est aussi $\mathcal{O}(n)$.
- On peut ici raisonnable supposer que l'accès à un élément de la liste ($L(i)$) est une opération élémentaire, i.e. en $\mathcal{O}(1)$;.

— Il y a 2 boucles imbriquées dans l'algorithme : le nombre *total* m d'itérations effectuées par la boucle intérieure est donné par :

$$m = 1 + 2 + 3 + \ldots + (n-1) = n(n-1)/2,$$

et pour chaque itération on effectue un nombre constant d'opérations.

Donc la complexité temporelle de l'algorithme est de la forme :

$$C_1 + C_2\, n + C_3\, \frac{n(n-1)}{2}.$$

où les C_i sont des constantes. Cet algorithme a donc une complexité en $\mathcal{O}(n^2)$.

Une méthode alternative consiste à d'abord trier les éléments de la liste, pour ensuite vérifier seulement les différences entre nombres consécutifs dans la liste triée :

La plus petite différence-2

entrée : *Liste L d'entiers contenant au moins 2 éléments*
sortie : *a, b nombres avec la plus petite différence*

```
L_triee = tri(L)
δ_min ⟵ ∞
t ⟵ taille(L_triee)
Pour i de 1 à t − 1
    δ ⟵ L_triee(i + 1) − L_triee(i)
    Si δ < δ_min
        δ_min ⟵ δ
        a ⟵ L_triee(i)
        b ⟵ L_triee(i + 1)
sortir : (a, b)
```

Dans ce cas, la complexité temporelle de l'algorithme dépend de celle de l'algorithme de **tri**, qui permet d'ordonner les éléments d'une liste.

En utilisant une approche naïve, la complexité temporelle d'un **tri** est $\mathcal{O}(n^2)$, et celle de notre algorithme est alors la même que pour la première version.

Des méthodes plus efficaces permettent de trier une liste de n nombres en $\mathcal{O}(n \log(n))$ opérations. La complexité temporelle de notre nouvel algorithme devient alors de la forme :

$$C_4 + C_5\, n + C_6\, n\, \log(n)$$

Et donc au final cet algorithme est en $\mathcal{O}(n \log(n))$; ce qui pour des listes de grande taille fait une grosse différence avec la première version !

7 PageRank

Après l'insertion d'un seul lien vers le blog de Rosa, son score PageRank, noté S_R, sera donné par l'équation

$$S_R = 0.15 + 0.85\, \frac{S}{L}$$

où S est le score PageRank de la page pointant vers le blog de Rosa, et L est le nombre de lien sortant de cette page.

Rosa doit donc choisir la page qui a le plus grand quotient S/L. Pour déterminer cela, il nous faut calculer le PageRank de chaque page web.

Commençons par écrire leurs équations. Soient S_G le score de George.com, S_F le score de Fabienne.fr, S_S le score de Sofien.ch et S_W le score de Wolfgang.me, et soit L_G le nombre de liens sortants de George.com, L_F le nombre de liens sortants de Fabienne.fr, etc. Nous avons alors :

$$L_G = L_S = 1 \qquad L_W = L_F = 2$$

et :

$$\begin{aligned} S_W &= 0.15 \\ S_G &= 0.15 + 0.85 \times (S_W/2 + S_F/2 + S_S) \\ S_F &= 0.15 + 0.85 \times S_G \\ S_S &= 0.15 + 0.85 \times (S_W/2 + S_F/2) \end{aligned}$$

Si l'on applique la méthode itérative de PageRank, on obtient :

	étape 1	étape 2	étape 3	étape 4	étape 5	...
S_W	0.25	0.15	0.15	0.15	0.15	...
S_G	0.25	0.57500	0.67594	0.79786	0.93412	...
S_F	0.25	0.36250	0.63875	0.72455	0.82818	...
S_S	0.25	0.36250	0.36781	0.48522	0.52168	...

Vous constaterez que l'on met du temps à converger vers la solution, ce qui rend le calcul assez rébarbatif...

Aparté

Si l'on veut être sûr, on pourrait ici résoudre les équations. **Ce n'est pas ce que nous vous demandons !**, mais nous vous l'indiquons pour information :

$$\begin{aligned} S_W &= 0.15 \\ S_G &= 0.15 + 0.85 \times (S_W/2 + S_F/2 + S_S) = 0.21375 + 0.425\,S_F + 0.85\,S_S \\ S_F &= 0.15 + 0.85 \times S_G \\ S_S &= 0.15 + 0.85 \times (S_W/2 + S_F/2) = 0.21375 + 0.425\,S_F \end{aligned}$$

et donc (soustraction de 2$^{\text{e}}$ et 4$^{\text{e}}$ équations)

$$S_G = S_S + 0.85 S_S = 1.85\,S_S$$

donc

$$S_F = 0.15 + 1.5725\,S_S$$

qui nous conduit à

$$S_S = 0.21375 + 0.063750 + 0.66831\,S_S$$

d'où

$$S_S = 0.83663 \qquad S_G = 1.54777 \qquad S_F = 1.46560$$

Nous vous avons demandé le calcul au bout de trois itérations. On a à ce stade :

$$S_S/L_S \simeq 0.36781, \qquad S_F/L_F \simeq 0.31938, \qquad S_W/L_W = 0.075, \qquad S_G/L_G \simeq 0.67594$$

Pour maximiser son score PageRank, Rosa devrait donc choisir de demander à George d'insérer un lien de George.com à son blog.

Si l'on avait fait le calcul exact (non demandé), nous aurions eu :

$$S_S/L_S \simeq 0.83663, \qquad S_F/L_F \simeq 0.77389, \qquad S_W/L_W = 0.075, \qquad S_G/L_G \simeq 1.54777$$

et serions arrivés à la même conclusion.

Pour aller plus loin

8 EdgeRank

Question 1. On constate que George a deux amis : Sofien et Fabienne. Si George se contente de poster sa nouvelle et que rien d'autre se passe, alors Rosa ne verra jamais la nouvelle car elle n'est pas amie avec George.

Question 2. Calculons le EdgeRank de la vidéo de George. Sofien a une affinité de 1 (faible) pour George. On obtient donc que le EdgeRank $E_S(\text{Vidéo})$ de la vidéo de George du point de Sofien est de $5 \times 1/h$, ou h est le nombre d'heures qui se sont écoulées depuis le « post » de George. Au bout de 10 heures, $E_S(\text{Vidéo})$ atteint 0.5 et la vidéo disparaît du fil d'actualité de Sofien.

Pour Fabienne, qui a une affinité forte pour George, on obtient $E_F(\text{Vidéo}) = 5 \times 2/h$. La vidéo sera donc visible sur le fil de Fabienne pendant 20 heures.

Question 3. Fabienne a deux amis qui ne sont pas amis de George alors que Sofien n'en a qu'un. Il faudra donc demander a Fabienne.

Question 4. Si Fabienne « like » le « post » de George, alors les deux amis de Fabienne, c'est-à-dire Tong et Rosa, pourront voir le « like » tant que son EdgeRank de leur point de bue sera supérieur à 0.5. Le EdgeRank du « like » de Fabienne du point de vue de Tong (noté $E_T(\text{Like}_F)$) sera supérieur à 0.5. Nous avons $E_T(\text{Like}_F) = 1 \times 2/h$. De même, pour Rosa, nous avons $E_R(\text{Like}_F) = 1 \times 1/h$. Si Fabienne « like » le « post » de George, alors Tong verra le « like » de Fabienne pendant 4 heures et Rosa pendant 2 heures.

Question 5. Supposons que Fabienne commente le « post » de George et que Tong « like » le commentaire de Fabienne une heure après. Soit h le temps écoulé depuis le commentaire de Fabienne. Notez que Tong et Fabienne sont tous les deux amis avec Rosa. Après le « like » de Tong, le score EdgeRank de la vidéo de George du point de vue de Rosa est donnée par la formule

$$E_R(\text{vidéo de George}) = 1 \times 3 \times 1/h + 2 \times 1 \times 1/(h-1) = 3/h + 2/(h-1)$$

La vidéo de George disparaîtra donc du fil de Rosa après environ 10 heures et demie ($h^2 - 11\,h + 6 = 0$, soit $h = \frac{11+\sqrt{97}}{2}$ car $h \geq 1$).

9 La plus courte séquence

1. Pseudo-code :

La plus courte séquence

entrée : *Liste M de `0` et de `1` ; $k \in \mathbb{N}$*
sortie : *longueur de la k^e plus courte séquence de 1 dans M*

```
lonSeq ⟵ vide
nouvSeq ⟵ true
j ⟵ 0
t ⟵ taille(M)
Pour i de 1 a t
    Si M[i] = 1
        Si nouvSeq = true
            lonSeq ⟵ lonSeq ∪ {1}
            j ⟵ j + 1
            nouvSeq ⟵ false
        Sinon
            lonSeq[j] ⟵ lonSeq[j] + 1
    Sinon
        nouvSeq ⟵ true
lonSeq ⟵ trier(lonSeq)
Si k > taille(lonSeq)
    afficher 0
Sinon
    afficher lonSeq[k]
```

2. Complexité. Le pire des cas pour l'algorithme donné se présente quand M est de la forme

$$(1, 0, 1, 0, 1 \ldots, 0, 1).$$

Soit n la longueur de M (dans notre cas impaire). Comme la complexité de l'algorithme **trier** pour une liste de longueur m est en $\mathcal{O}(m \log(m))$ et que le calcul de la taille d'une liste de longueur m est *au pire* en $\mathcal{O}(m)$, l'ordre de grandeur du nombre d'opérations pour chaque ligne de l'algorithme, indiquée en rouge ci-dessous, est :

La plus courte séquence (complexité)

entrée : *Liste M de `0` et de `1` ; $k \in \mathbb{N}$*
sortie : *longueur de la k^e plus courte séquence de 1 dans M*

```
lonSeq ⟵ vide                                   1
nouvSeq ⟵ true                                  1
j ⟵ 0                                           1
t ⟵ taille(M)                                   O(n)
Pour i de 1 a t                                 2
    Si M[i] = 1                                 2
        Si nouvSeq = true                       1
            lonSeq ⟵ lonSeq ∪ {1}               2
            j ⟵ j + 1                           2
            nouvSeq ⟵ false                     1
        Sinon
            lonSeq[j] ⟵ lonSeq[j] + 1           2 (n'est pas executé)
    Sinon
        nouvSeq ⟵ true                          1

lonSeq ⟵ trier(lonSeq)                          O((n+1)/2 log((n+1)/2))
Si k > taille(lonSeq)                           O((n+1)/2)
    afficher 0                                  1 (n'est pas executé)
Sinon
    afficher lonSeq[k]                          2
```

La complexité de l'algorithme est alors :

$$\begin{aligned} C(n) &\in \mathcal{O}\left(4 + n + 4 \cdot n + 6 \cdot \frac{n+1}{2} + 1 \cdot \frac{n-1}{2} + 1 \right. \\ &\qquad \left. + \frac{n+1}{2} \log\left(\frac{n+1}{2}\right) + 1 + \left(\frac{n+1}{2}\right) + 2\right) \\ &\in \mathcal{O}\left(n + n \log\left(n+1\right) + \log\left(n+1\right)\right) \\ &\in \mathcal{O}\left(n \log n\right). \end{aligned}$$

Pour le fun...

Un algorithme de tri inhabituel

Quand il sort de la caverne, chaque nain peut voir la couleur des bonnets de ses compagnons qui sont déjà sortis. Si tous les nains déjà à l'extérieur de la caverne ont le bonnet de la même couleur, alors il va se placer à une extrémité de la ligne.

Si non, supposons que les autres nains soient déjà rangés dans le bon ordre, avec (par exemple) tous les bonnets rouges à droite et les bonnets bleus à gauche. Alors, il suffit que le nain se place à l'endroit du changement de couleur de bonnet, entre le dernier nain avec un bonnet rouge et le premier avec un bonnet bleu. Sauriez-vous démontrer que cette solution est correcte ?

```
....RRRRRRR BBBBBBBBB......
           ^
           |
           X
```

Une variante

Voici une solution possible : soit r_1 le nombre donné par le magicien au premier nain. Si ce nombre est pair, il annonce « bleu » (« 0 »), tandis que si ce nombre est impair, il annonce « rouge ». Ce chiffre n'a bien sûr rien à voir a priori avec la couleur de son propre bonnet : il n'a donc que 50% de chances de s'en sortir. Mais les suivants ont maintenant une information précieuse ! Le suivant qui sort compte le nombre de bonnets rouges que lui dit le magicien : appelons ce nombre r_2. Il y a alors 2 possibilités :

- soit r_1 et r_2 ont la même parité (tous les deux pairs ou tous les deux impairs ; il connaît la parité de r_1 grâce au message du nain précédent) : vu que la différence entre r_1 et r_2 ne peut pas être plus grande que 1, le nain sortant sait que $r_1 = r_2$, i.e., que le nombre de bonnets rouges n'a pas changé, et donc que son propre bonnet est forcément bleu : il annonce donc « bleu » et s'en sort ;
- soit r_1 et r_2 sont de parité contraire : dans ce cas, on a forcément que $r_2 = r_1 - 1$, et donc le nain sortant sait que son propre bonnet est rouge ; il annonce donc « rouge »et s'en sort également.

Remarquez que dans tous les cas, ce que le 2^{e} nain annonce avec cette méthode n'est en fait que la différence de parité entre r_1 et r_2 : « 0 » (« bleu ») si r_1 et r_2 ont la même parité et « 1 » s'ils ont une parité différente.

Que fait alors le 3e nain ?
D'abord, il déduit des deux premières annonces si le nombre de bonnets rouges restants (inclus le sien) est pair ou impair :

- s'il a été annoncé « rouge, rouge (1,1) » ou « bleu, bleu (0,0) », il sait que le nombre de bonnets rouges restants est pair ;
- s'il a été annoncé « rouge, bleu (1,0) » ou « bleu, rouge (0,1) », il sait que le nombre de bonnets rouges restants est impair.

Il compte ensuite le nombre de bonnets rouges que lui dit le magicien, qu'on appellera r_3. Si ce nombre a la même parité que ce qu'il vient de calculer, il sait que son propre bonnet est bleu, et il annonce « bleu ». Sinon, il annonce « rouge ».

Et ainsi de suite jusqu'au dernier : ainsi, tout le monde est sauf (excepté le premier, à 50%).