IFT2015 automne 2016 - Devoir1

Philippe Caron

19 septembre 2016

1.1 Chemins sur une grille

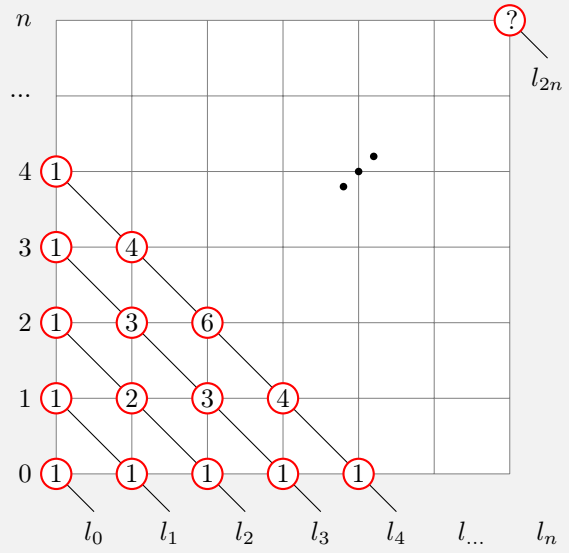
a.

On cherche à trouver le nombre de chemin qui se rendent à un point (n, n) en partant de $(0, 0)$ en ne pouvant que monter (incrémenter y) et aller à droite (incrémenter x).

Soit $N_{x,y}$ le nombre de chemin qui se rendent au point (x, y) tel que $x, y \in \mathbb{N}$.

On trouve le nombre de chemin à chaque nœud dans la grille grâce à la fonction récursive suivante :

$$N_{x,y} = \begin{cases} 1 & x = 0, y = 0 \\ N_{0,y-1} & x = 0, y \geq 1 \\ N_{x-1,0} & x \geq 1, y = 0 \\ N_{x-1,y} + N_{x,y-1} & x \geq 1, y \geq 1 \end{cases} \tag{1}$$



En calculant la valeur de chaque nœud en utilisant la fonction récursive définie à l'équation (1), on remarque très rapidement le développement du triangle de Pascal, ce qui nous donne un bon indice sur la manière de procéder.

Tous les chemins menant à un même nœud sont forcément de la même taille puisque la position dépend directement du nombre de déplacement, mais l'ordre dans lequel ces déplacement sont faits n'a pas d'importance. Appelons $\vec{i}$ et $\vec{j}$ les vecteurs unitaires croissants et parallèles à l'axe x et l'axe y respectivement tels que :

$$l_{x,y} = ||x\vec{i} + y\vec{j}||$$

Où $l_{x,y}$ est la longueur de tous les chemins se rendant à $N_{x,y}$.

Pour obtenir le nombre de chemin, il suffit de trouver tous les réarangement possibles de $\vec{i}$ et $\vec{j}$. Prenons par exemple le point $(1, 2)$ sur le tableau :

$$\begin{aligned} l_{1,2} &= ||\vec{i} + \vec{j} + \vec{j}|| \\ &= ||\vec{j} + \vec{i} + \vec{j}|| \\ &= ||\vec{j} + \vec{j} + \vec{i}|| \end{aligned}$$

On conclu que :

$$N_{1,2} = 3$$

De manière générale, il y a $k!$ façon d'arranger k éléments distincts. Puisque dans le cas qui nous intéresse les $\vec{i}$ et $\vec{j}$ sont indifférenciables, il faut enlever à ce nombre toutes les permutations composées strictement de $\vec{i}$ ou de $\vec{j}$. Ce qui nous conduit à la règle générale suivante :

$$N_{x,y} = \frac{(x+y)!}{x! \cdot y!} = \binom{x+y}{x} = \binom{x+y}{y} \tag{2}$$

Il s'en suit donc que le nombre de chemin possible dans une grille de taille n est donné par :

$$C(n) = N_{n,n} = \binom{2n}{n} = \frac{(2n)!}{n! \cdot n!} \tag{3}$$

CQFD

b.

Afin de trouver une fonction simple asymptotiquement égale à $\ln C(n)$, substituons d'abord toutes les occurences de factorielle dans $C(n)$ par la formule de stirling.

$$\begin{aligned}
\frac{(2n)!}{n! \cdot n!} &\overset{\text{Stirling}}{\sim} \frac{\sqrt{2\pi(2n)}\left(\frac{(2n)}{e}\right)^{(2n)}}{\left(\sqrt{2\pi n}\left(\frac{n}{e}\right)^{n}\right)^{2}} \text{ (quand } n \to \infty) \\
&= \frac{2\sqrt{\pi n}\left(\frac{2n}{e}\right)^{2n}}{2\pi n\left(\frac{n}{e}\right)^{2n}} \\
&= \frac{2^{2n}}{\sqrt{\pi n}} \\
&= \frac{4^n}{\sqrt{\pi n}}
\end{aligned} \tag{4}$$

Puis nous pouvons rajouter le logarithme naturel :

$$\ln\left(\frac{4^n}{\sqrt{\pi n}}\right) = \ln(4) \cdot n - \frac{1}{2} \cdot \ln(\pi n) \tag{5}$$

Dans l'équation (5), le terme avant la soustraction grandi plus vite, on fait donc l'hypothèse que la fonction la plus simple asymptotiquement égale à $\ln C(n)$ est $f(n) = \ln(4) \cdot n$, on fait le test des limites pour vérifier : *

$$\begin{aligned}
\lim_{n\to+\infty} \frac{\ln C(n)}{f(n)} &\overset{5}{=} \frac{\ln\left(\frac{4^n}{\sqrt{\pi n}}\right)}{\ln(4) \cdot n} \\
&\overset{4}{=} \frac{\ln(4) \cdot n - \frac{1}{2} \cdot \ln(\pi n)}{\ln(4) \cdot n} \\
&\overset{\text{RH}}{=} \frac{\left(\ln(4) \cdot n - \frac{1}{2} \cdot \ln(\pi n)\right)'}{\left(\ln(4) \cdot n\right)'} \\
&= \frac{\ln(4) - \frac{1}{2\pi n}}{\ln(4)} \\
&= \frac{\ln(4) - \cancelto{0}{\frac{1}{2\pi n}}}{\ln(4)} \\
&= \frac{\ln(4)}{\ln(4)} \\
&= 1
\end{aligned} \tag{6}$$

∗. RH = Règle de l'Hôpital

Nous avons ainsi prouvé que $\ln C(n) \sim f(n)$ (quand $n \to \infty$) où :

$$f(n) = \ln(4^n) = n \cdot \ln(4)$$

1.2 Comment encoder les entiers ?

a.

Les valeurs sont les suivantes :

$$|\mathbf{u}(10^{100})| = 10^{100} + 1$$
$$|\mathbf{b}(10^{100})| = 333$$
$$|\mathbf{f}(10^{100})| = 667$$
$$|\mathbf{g_1}(10^{100})| = 350$$

De manière générale :

$$|\mathbf{u}(n)| = n + 1$$
$$|\mathbf{b}(n)| = \lg(n+1)$$
$$|\mathbf{f}(n)| = 2 \cdot \lg(n+1) + 1$$

b.

Voici le pseudo-code pour l'encodage de $\mathbf{b}(n)$, où «/» est la division entière, l'algorithme est facile à implémenter en python en utilisant, les strings, il suffit de remplacer n par $str(n)$ à la dernière ligne.

```
EncodeBeta(n) :
    if n > 1 then :
        return EncodeBeta(n/2) ⊕ modulo(n, 2)
    else :
        return n
```

c.

Voici le pseudo-code pour l'encodage de $\mathbf{g_1}(n)$

```
EncodeGamma(n) :
    if n == 0 then :
        return ""
    else :
        l ← EncodeBeta(n)
        return EncodeGamma(length(l) − 1) ⊕ l


EncodeOmega(n) :
    return EncodeGamma(n) ⊕"0"
```

Si on est vraiment attaché à ce que la fonction EncodeOmega soit purement récursive, on peut utiliser une variable booléenne dans les paramètres et conditionnellement ajouter un zéro à la fin de la chaîne, puis faire l'appel récursif avec «faux». Cependant, en plus d'être très désagréable à utiliser, cette fonction effectue des tests inutiles à chaque itération.

Voici le code en Python :

```
def encodeBeta(n):
   if n > 1:
      return encodeBeta(n // 2) + str(n % 2)
   else:
      return str(n)

def encodeGamma(n):
   if n == 0:
      return ''
   else:
      l = encodeBeta(n)
      return encodeGamma(len(l) - 1) + l

def encodeOmega(n):
   return encodeGamma(n) + '0'
```

d. Bonus

Pour sauver du temps, la fonction $\mathbf{g_1}$ n'a pas été implémentée, mais on suppose qu'elle fonctionne comme suit :

— Elle lit tous les mots du documents et les ajoute à une liste qui compte le nombre d'ajout
— Elle écris la liste au début du fichier ordonnée par fréquence
— Elle défini la longeur des fragments
— Elle écrit à la place de chaque mot un fragment de taille constante contenant l'index dans la liste

On considère donc un mot (séparé d'espace) comme étant un symbole. Ainsi pour avoir le ratio de compression il suffit d'avoir la taille de la liste au début (l'ordonner ne change pas la taille), le nombre de mots dans le texte (qui deviendrons des fragments) et le nombre d'octets dans le document original.

On détermine la taille minimale des fragments en calculant le logarithme du nombre d'élément dans la liste (ce qui donne le nombre de bit minimum pour définir chacun des index), puis en divisant celui-ci par 8 pour obtenir le nombre d'octets nécessaire.

Bien entendu, la fonction $\mathbf{g_1}$ telle que définie présentement laisse beaucoup à désirer et serait très facile à optimiser, le code qui examine le ratio de compression est lui aussi très sommaire et pourrait être plus rigoureux, mais vu que le but n'est que d'étudier l'efficacité de la méthode, toute ces améliorations seraient fort coûteuses pour peu de résultat.

Les textes choisi pour l'étude sont «Moby Dick», «War and Peace» et «World 192», et ils ont été comparés à la compression zip, voici les résultats :

```
mobydick.txt:
Il y avait 1191463 dans le fichier,
il y a maintenant 666794.
Ratio de compression : 0.44035693932585396
zip: 59%

world192.txt:
Il y avait 2473400 dans le fichier,
il y a maintenant 1162955.
Ratio de compression : 0.5298152340907254
zip: 71%

war+peace.txt:
Il y avait 3202941 dans le fichier,
il y a maintenant 1458212.
Ratio de compression : 0.5447271741814788
zip: 63%
```

Comme on peut voir, cette méthode est assez efficace ! Même si elle l'est environ 15% moins que le zip à peu près, elle donne tout de même un résultat constant d'environ 50% à chaque fois.

Il est important de noter cependant que cela ne fonctionne qu'avec les textes fragmentés par des espaces, puisque toute la méthode repose entièrement là dessus. Sur des documents binaire, ou de code génétique, la compression résulterait en un plus gros fichier !