Burning Coins

Guillaume Bequet

7 janvier 2023

1 Solution récursive

Soit V le tableau contenant les valeurs v_i des n pièces.

On définit la fonction récursive *burningCoinsRec* prenant en paramètres deux entiers i et j ainsi que le tableau V et renvoyant le gain garantie dans la configuration où il ne reste que les pièces allant de i à j.

$$burningCoinsRec(i,j,V) = \begin{cases} 0 & \text{si} i > j \\ v_i & \text{si i = j} \\ max(v_i, v_j) & \text{si i = j-1} \\ max \begin{cases} min \begin{cases} v_i + burningCoinsRec(i+2,j,V) \\ v_i + burningCoinsRec(i+1,j-1,V) \end{cases} \\ min \begin{cases} v_j + burningCoinsRec(i,j-2,V) \\ v_j + burningCoinsRec(i+1,j-1,V) \end{cases} \end{cases} & \text{sinon} \end{cases} \quad (1)$$

L'appel initial est : *burningCoinsRec(1, n, V)*.

Il existe 3 cas d'arrêts :
— Si les indices ne sont pas cohérents ($i > j$), on renvoi 0.
— S'il ne reste qu'une pièce ($i = j$), on renvoi v_i.
— S'il reste seulement 2 pièces ($i = j - 1$), on renvoi $max(v_i, v_j)$.

Pour le cas général on essaie de maximiser le gains sans tenir compte de la stratégie de l'adversaire. Nous avons le choix entre prendre la pièce à l'extrémité gauche ou celle à l'extrémité droite. Notre adversaire aura les mêmes choix ensuite. Par exemple si l'on prend la $i_{ème}$ pièce, pour simuler le fait que l'adversaire prendra les décisions réduisant au maximum nos gains on regarde le minimum selon s'il prend la $(i+1)_{ème}$ pièce ou la $j_{ème}$. Le raisonnement est symétrique si notre choix se porte sur la $j_{ème}$ pièce. On renvoi alors le maximum entre ces deux valeurs.

1.1 Complexité

La solution récursive semble avoir l'avantage de n'utiliser aucune mémoire. Cependant la pile d'exécution peut croître exponentiellement dû aux nombreux appels récursif.

Pour ce qui est de la complexité temporelle, nous allons nous intéresser à la borne asymptotique inférieure Ω. Calculons la complexité dans le cas général (le quatrième cas de l'équation (1)) :

$$\begin{aligned} T(1,n) &= T(3,n) + T(2,n-1) + T(1,n-2) + T(2,n-1) \\ &\geq 2 \times T(2,n-1) \end{aligned} \quad (2)$$

Comme le cas d'arrêt se produit lorsque $i = j$ on trouve donc que $T(1,n) \in \Omega(2^{n/2})$.

2 Solution utilisant la programmation dynamique

La première étape pour définir un programme réduisant la complexité temporelle et utilisant le paradigme de programmation dynamique est de dessiner le graphe de dépendance. Les figures 1 et 2 montre des exemples de graphe de dépendance selon la parité du nombre de pièce.

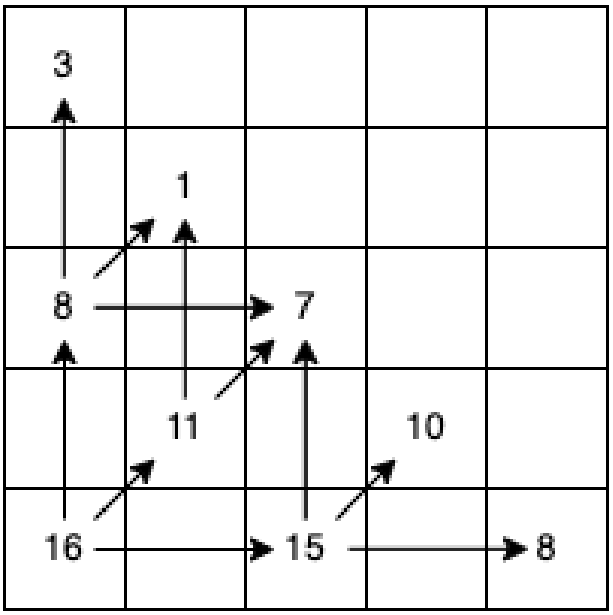

FIGURE 1 – Tableau rempli pour $V = [3, 1, 7, 10, 8]$

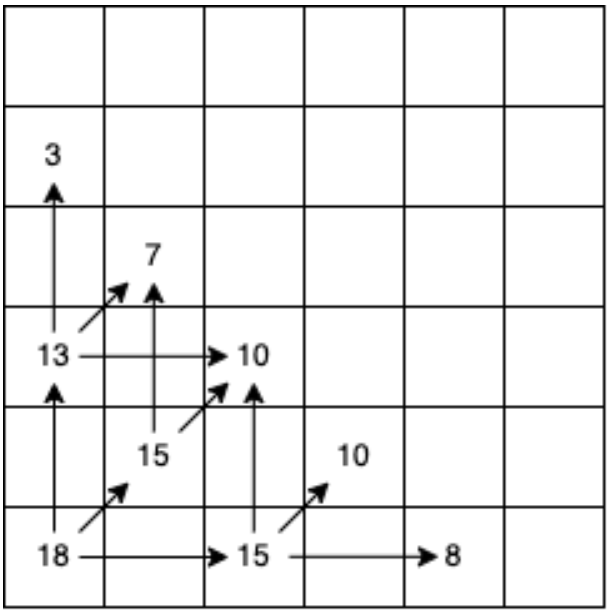

FIGURE 2 – Tableau rempli pour $V = [3, 1, 7, 10, 8, 1]$

La diagonale dépend de la parité du nombre de pièce, si n est impair lors de notre dernier tour il ne restera qu'une pièce, sinon nous aurons le choix entre deux pièces. On peut donc remplir la diagonale au début.

On parcourt ensuite les diagonales parallèles en remplissant les cases selon la formule définit dans le quatrième cas de l'équation (1). À la fin du programme le résultat se trouve dans la case $[1, n]$.

2.1 Complexité

On utilise un tableau $n \times n$ est alloué pour calculer le résultat, la complexité mémoire est donc en $\Theta(n^2)$.

Pour la complexité temporelle on itère les calculs sur les $n/2$ diagonales. La taille de la $i_{ème}$ diagonale est $n - 2i$. La fonction $T(n)$ donnant la complexité de l'algorithme pour un tableau de taille n est la suivante :

$$\begin{aligned} T(n) &= \sum_{i=1}^{n/2} n - 2i \\ &= \sum_{i=1}^{n/2} n - 2\sum_{i=1}^{n/2} i \\ &= \frac{n^2}{2} - \frac{n}{2}(\frac{n}{2} + 1) \\ &= \frac{n^2}{2} - \frac{n^2 + 2n}{4} \\ &= \frac{n^2 - 2n}{4} \end{aligned} \tag{3}$$

On a donc $T(n) \in \Theta(n^2)$.

Algorithm 1 Résolution burning coins avec tableau deux dimensions, `BurningCoins2D`

input : Valeurs des pièces, $V = \{v_1, v_2, ..., v_n\}$,
Nombre de pièces, n.
output: Le gain garantie indépendamment de la stratégie de l'adversaire

allouer un tableau T de taille $n \times n$
for i *de* 1 *à* $n-1$ **do**
  $T[i,i] \leftarrow v[i]$
  $T[i+1,i] \leftarrow max(v[i], v[i+1])$
end
$T[n,n] \leftarrow v[n]$
if n *est impair* **then**
  $p \leftarrow 1$
else
  $p \leftarrow 2$
end
for i *de* p *à* n *avec un pas de 2* **do**
  for j *de 1 à* $n-i$ **do**
    $k \leftarrow i+j$
    $$T[k,j] \leftarrow max \begin{cases} min \begin{cases} v[k] + T[k-1, j+1] \\ v[k] + T[k-2, j] \end{cases} \\ min \begin{cases} v[j] + T[k-1, j+1] \\ v[j] + T[k, j-2] \end{cases} \end{cases}$$
  end
end
return $T[1,n]$

3 Solution réduisant l'espace mémoire utilisé

La solution précédente à le défaut d'allouer $n \times n$ cases mémoires alors que beaucoup ne sont pas utilisées. Aussi certaines ne seront utiles qu'à un certains moment du calcul ; par exemple dans la figure 1 la case $[1,1]$ n'est utilisée que pour le calcul de la case $[3,1]$.

On peut alors réduire la complexité mémoire de l'algorithme en utilisant un tableau de taille n et en écrasant les résultats inutiles pour le reste du calcul (voir algorithme 2).

Ceci rend la complexité mémoire linéaire ($\Theta(n)$). La complexité temporelle reste quant à elle la même (ie. $\Theta(n^2)$).

4 Implémentation

J'ai d'abord décidé d'implémenter la solution en Python car j'ai cru comprendre que c'était le langage le plus utilisé par l'équipe. Cependant le temps d'exécution me paraissait long sur ma machine (plus de 3 minutes pour le troisième lot de tests).

Je me suis décidé à le refaire en C et le temps d'exécution à littéralement était divisé par 600.

Les arguments sont toutefois les mêmes :
— En Python
`burning_coins.py chemin_fichier_entrée chemin_fichier_sortie`
— En C
```
gcc burning_coins.c -o exe
./exe chemin_fichier_entrée chemin_fichier_sortie
```
Pour le code C j'ai pré compilé un exécutable si jamais.

Le lien du Git : https://github.com/lommick1/burning_coins.

Algorithm 2 Résolution burning coins avec tableau une dimension, `BurningCoins1D`

input : Valeurs des pièces, $V = \{v_1, v_2, ..., v_n\}$,
Nombre de pièces, n.

output: Le gain garantie indépendamment de la stratégie de l'adversaire

19 allouer un tableau T de taille n
20 **if** *n est impair* **then**
21 $p \leftarrow 1$
for *i de 1 à n* **do**
22 $T[i] \leftarrow v[i]$
23 **end**
24 **else**
25 $p \leftarrow 2$
for *i de 1 à $n-1$* **do**
26 $T[i] \leftarrow max(v[i], v[i+1])$
27 **end**
28 **end**
29 **for** *i de p à n avec un pas de 2* **do**
30 **for** *j de 1 à $n-i$* **do**
31 $k \leftarrow i + j$
32 $$T[j] \leftarrow max \begin{cases} min \begin{cases} v[k] + T[j] \\ v[k] + T[j+1] \end{cases} \\ min \begin{cases} v[j] + T[j+1] \\ v[j] + T[j+2] \end{cases} \end{cases}$$
33 **end**
34 **end**
35 **return** $T[1]$