PROGRAMMATION DYNAMIQUE

1 Un exemple simple et classique

Traditionnellement, ce problème s'appelle rod-cutting *et parle de barres de métal. Je vous livre ici une version chocolatée écrite pour mes élèves, après tout c'est Pâques...*

Après un changement de l'équipe dirigeante, l'entreprise Mondelez International, qui fabrique les barres Toblerone, décide de rationaliser sa production pour maximiser ses revenus. En effet, leur chaîne de production fabrique des barres de n «carreaux» qui sont ensuite coupées en barres plus petites avant d'être vendues. Mais une récente étude de marché a déterminé le prix auquel on pouvait vendre des barres de longueur k (pour $1 \leqslant k \leqslant n$), et ce prix s'avère ne pas avoir de relation simple avec k. Le problème est donc de décider comment découper la barre initiale de n carreaux en des barres plus petites pour maximiser le prix de vente total. Pour simplifier, on néglige le coût de la découpe et de l'emballage.

Dans tout le problème, on considérera que l'on dispose d'un tableau[1] `tab_prix`, indicé de 0 à $n-1$, tel que `tab_prix[k]` soit le prix de vente d'une barre de longueur $k+1$. Remarquez que si `tab_prix[n - 1]` est suffisamment grand, la solution optimale peut très bien être de ne pas découper la barre.

Le cœur du problème.

On donne un exemple de tableau t pour $n = 10$. Attention, j'indice les t_i de 1 jusqu'à n (c'est plus naturel), mais `tab_prix` est bien sûr indicé de 0 à $n-1$.

i	1	2	3	4	5	6	7	8	9	10
t_i	1	5	8	9	10	17	17	20	24	26

La solution optimale pour cet exemple est de découper la barre en deux morceaux de taille 2 et un morceau de taille 6, pour une valeur totale de 27.

1.1 Solution itérative naïve

La solution la plus simple est d'énumérer toutes les découpes possibles et de déterminer celle ayant la plus grande valeur. Il y a quand même une petite difficulté : qu'est-ce qu'une découpe ? Logiquement, on ne devrait pas tenir compte de l'ordre des morceaux : pour une barre de taille 10, les découpes 3, 3, 2, 2 et 2, 3, 2, 3, par exemple, sont clairement équivalentes. On pourrait donc décider d'une représentation canonique (du plus petit au plus grand morceau, disons) et tenter d'énumérer ces représentations. C'est possible (et c'est en gros ce que l'on fera dans la

1. c'est-à-dire d'une `list` python.

solution récursive naïve), mais ce n'est pas complètement évident. On va donc aller au plus simple et utiliser un codage redondant : à un tableau d de $n-1$ booléens, on peut associer la découpe dans laquelle on a coupé après le $k+1$-ème morceau si et seulement si d_k est vrai.

Exercice 1 **À traiter plus tard**

1. *Que peut-on attendre comme complexité pour une solution* brute force *au problème ?*
2. *Implémenter deux variantes* brute force *: en générant récursivement la liste des listes de booléens de taille $n-1$, ou en bouclant pour k de 0 à $2^{n-1}-1$ et en calculant la représentation binaire de k.*

1.2 Sous-structure optimale : solution récursive

Pour arriver à une solution efficace, la première étape est de remarquer qu'une solution du problème (*i.e.* une découpe optimale d'une barre de taille n) «contient» des solutions à des versions plus petites du même problème. Pour simplifier, on peut supposer (sans perte de généralité), qu'à chaque étape on découpe ce qui reste de la barre en un morceau de taille k ($1 \leqslant k \leqslant n$) qui fera partie de la découpe finale et une nouvelle barre qui pourra à nouveau être découpée. Il est alors clair que cette nouvelle barre (de longueur $n-k$) devra elle aussi être découpée de manière optimale. En notant $v(n)$ la valeur optimale d'une découpe d'une barre de taille n et en posant $v(0)=0$, on obtient [2] :

$$v(n) = \max_{1 \leqslant k \leqslant n} (t_k + v(n-k))$$

Exercice 2

Écrire une fonction récursive `val_opt_rec(tab_prix, n)` *exploitant «bêtement» la relation ci-dessus.*
Si l'on veut respecter la signature de départ (et donc ne pas avoir n comme deuxième argument, il suffit évidemment de faire un peu d'emballage : on renomme `val_opt_rec(tab_prix, n)` en `val_opt_aux(tab_prix, n)` et on écrit une fonction `val_opt_rec(tab_prix)` qui ne fait qu'appeler `val_opt_rec(tab_prix, len(tab_prix))`.

1.3 Analyse de la solution récursive naïve

Un joli dessin valant mieux qu'un long discours, voilà quelques exemples d'arbres d'appels pour la version récursive basique présentée plus haut. Vu qu'ils deviennent rapidement gros, j'ai omis les valeurs.

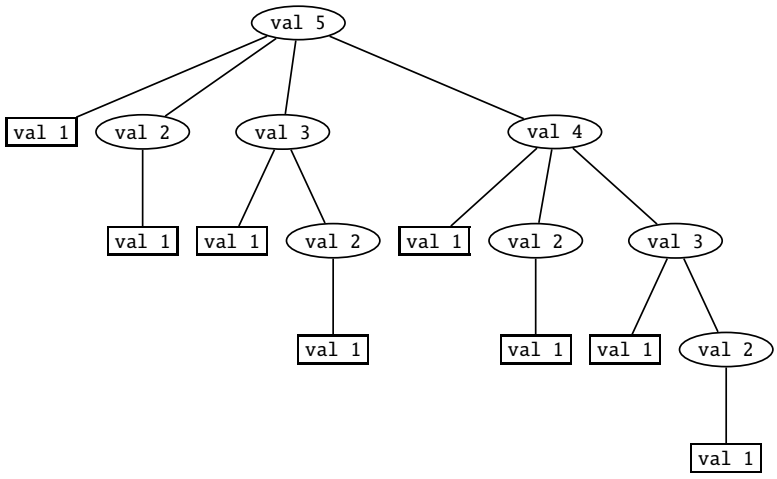

Figure 1 – Découpage de toblerone pour $n=5$, récursif naïf

L'arbre se lit comme suit :
- on appelle `val-opt_rec(5)` (noté simplement `val 5` ici) ;
- `val 5` appelle successivement `val 1`, `val 2`, `val 3` et `val 4` ;

2. de nombreuses variations sur cette formule sont possibles, en particulier si l'on s'oblige à découper le plus petit morceau en premier.

- `val 1` renvoie immédiatement (feuille) ;
- `val 2` appelle `val 1` avant de renvoyer ;
- `val 3` appelle `val 2`, qui appelle…
- …
- `val 4` renvoie, et `val 5` peut maintenant renvoyer également.

Il y a juste une chose par forcément évidente à comprendre sur ce type d'arbre : les appels actifs à un moment donné sont tous ceux situés sur le chemin reliant le nœud actuel à la racine. Ceux situés plus à gauche (quel que soit le niveau) sont déjà terminés, alors que ceux situés plus à droite n'ont pas encore commencé. Un bon exercice pour être sûr que l'on a compris est d'essayer de tracer l'arbre correspondant à la version mémoïsé avant de comparer avec celui que l'on obtient effectivement.

Pour $n = 6$, ça tient encore sur la page, en serrant bien :

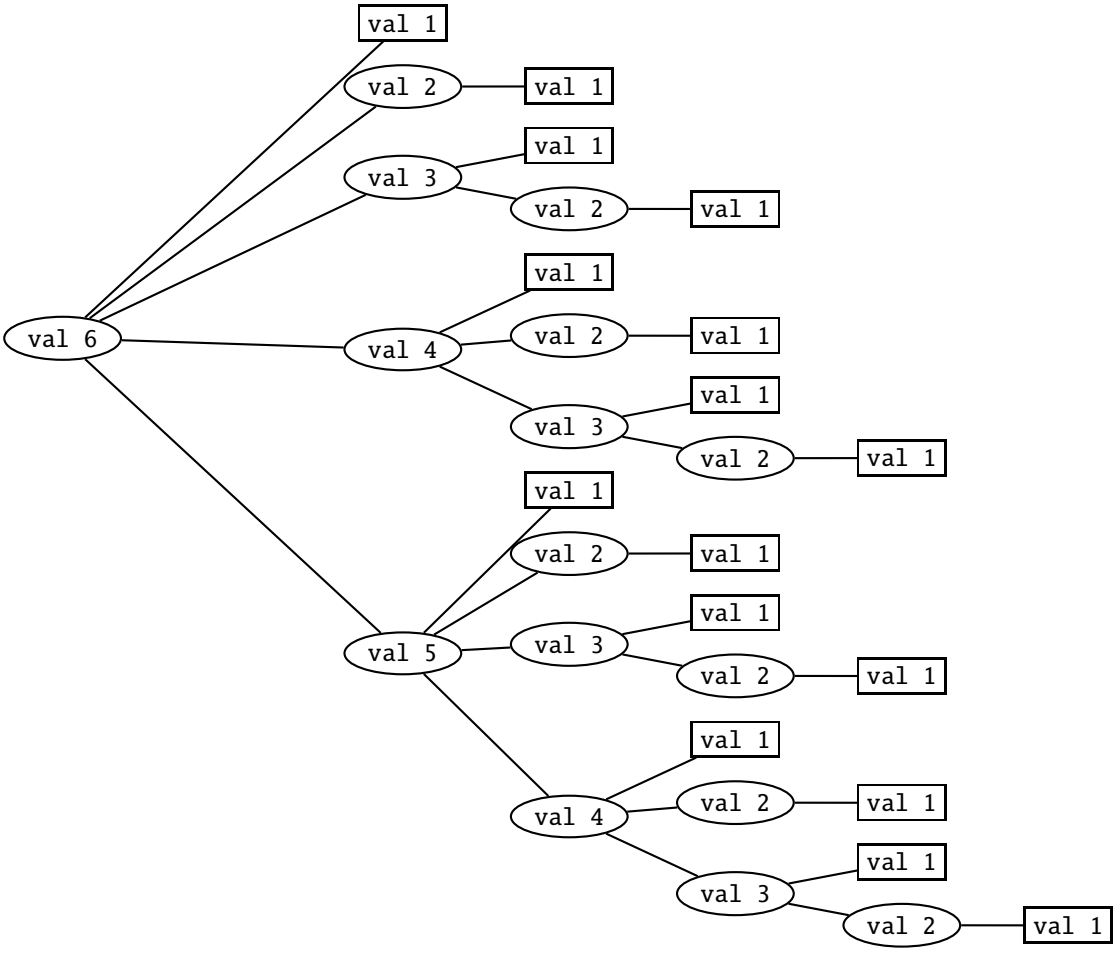

Figure 2 – Découpage de toblerone pour $n = 6$, récursif naïf

Le problème est immédiatement apparent : on passe notre temps à recalculer la même chose ! C'est précisément dans les cas où les sous-problèmes se chevauchent (et donnent donc lieu à une répétition de calculs) que la programmation dynamique est intéressante.

1.4 Solution dynamique : version itérative

Une manière plus «intelligente» d'utiliser la relation de récurrence que l'on a trouvé est de construire un tableau auxiliaire contenant les v_k. On l'initialise à 0, on le remplit au fur et à mesure [3], et quand on a terminé on récupère le dernier élément (qui correspond à $v(n)$, la valeur cherchée).

Exercice 3

3. ou alors on le fait grandir à coups d'`append`

Implémenter la solution...

1.5 Solution dynamique : récursion avec mémoïsation

En fait, il est tout aussi simple de garder une solution récursive. Il suffit que la fonction vérifie si la solution a déjà été calculée. Si c'est le cas, on renvoie directement le résultat, sinon on le calcule, **puis on le stocke**, puis on le renvoie. Ce principe porte le nom de *mémoïsation*[4] Il faut une mémoire globale (un cache) partagée par tous les appels de la fonction, qui doit donc être créée à l'extérieur de celle-ci. Typiquement, on se retrouve avec un enrobage : le seul rôle de la fonction extérieure est de créer le cache initial et de passer la main à la fonction intérieure (récursive et mémoïsée) qui fait le travail.

EXERCICE 4

1. *Écrire une fonction* `val_opt_mem_aux(n, tab_prix, tab_aux)` *qui procède comme suit :*
   - *si* `tab_aux[n]` *a une «vraie» valeur (pas* -1*, par exemple), on la renvoie ;*
   - *sinon, à l'aide de la relation de récurrence vue plus haut, on la calcule, puis on la stocke dans le tableau* `tab_aux[n]` *(ou* $n-1$*, à vous de voir), puis on la renvoie.*
2. *Écrire alors une fonction* `val_opt_mem(n, tab_prix)` *qui réponde au problème.*
3. *Quelle est la complexité de cette fonction ?*

On peut à nouveau regarder ce que donnent les arbres d'appels : c'est beaucoup plus raisonnable !

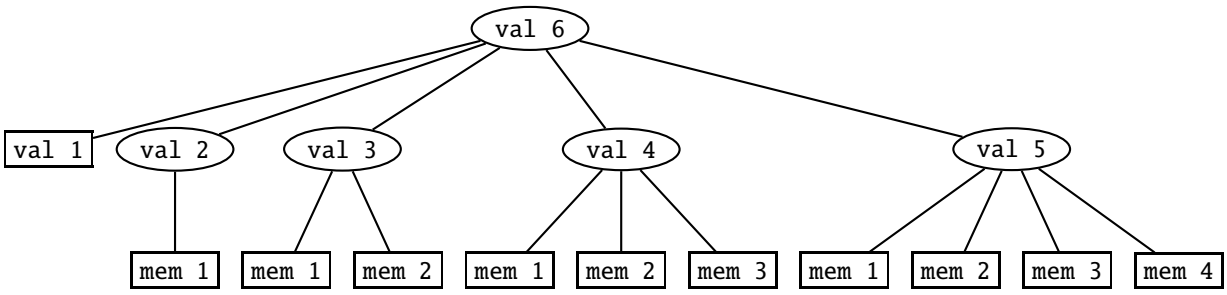

FIGURE 3 – Découpage de toblerone pour $n = 6$, récursif mémoïsé.

1.6 Reconstruction de la découpe optimale

Pour l'instant, on a déterminé la valeur optimale d'une découpe de la barre, mais on n'est pas en mesure d'exhiber une découpe optimale. C'est à peu près systématique dans ce genre de problème : on commence par se préoccupper de déterminer la valeur optimale, puis on regarde ce qu'il faut modifier à l'algorithme pour reconstruire la solution. Typiquement, on est amené à stocker un peu d'information supplémentaire à chaque étape, puis, une fois qu'on a fini le calcul de la valeur optimale, à reconstruire les choix faits étape par étape.

Ici, on peut par exemple stocker dans `tab_aux[k]` le couple (i, v_k), où v_k est la valeur optimale pour k et i est la longueur d'un morceau[5] final (*i.e.* qui ne sera pas redécoupé) de la décomposition optimale pour k. Dans ce cas, une fois que l'on a fini de remplir `tab_aux`, on peut aller lire son dernier élément `tab_aux[-1]` : s'il vaut $(2, 27)$, par exemple, on sait qu'on a un 2 dans notre découpe et on va aller lire `tab_aux[- 1 - 2]` pour continuer.

EXERCICE 5

Le faire ! Plus précisément, écrire une fonction `opt_dyn(tab_prix)` *ou* `opt_mem(tab_prix)` *(au choix) qui renvoie une découpe optimale, en se basant soit sur la méthode dynamique itérative, soit sur la méthode dynamique par récursion mémoïsée.*
On représentera la découpe par la liste des morceaux qu'elle contient (vous l'obtiendrez sans doute triée par ordre croissant, mais ça n'a pas vraiment d'importance).

4. Uniquement pour embêter les correcteurs orthographiques ?
5. le plus petit typiquement, mais ce n'est pas important.

2 Considérations générales

2.1 Classe de problèmes concernés

Au départ, la programmation dynamique s'applique à des problèmes d'optimisations ayant les deux propriétés suivantes :

- *sous-structure optimale* : une solution contient des solution à des instances plus petites du même problème ;
- *chevauchement des sous-problèmes* : une solution récursive en suivant le principe du «diviser pour régner» conduit à résoudre plusieurs fois les mêmes sous-problèmes.

Notons que les idées de la programmation dynamique peuvent être utilisées dans un cadre un peu plus large : on verra plus loin des exemples qui ne sont pas des problèmes d'optimisation. Le plus souvent, la difficulté est de transformer le problème initial (typiquement en calculant un peu plus que ce qui est demandé) pour que la propriété de sous-structure optimale soit vérifiée.

2.2 Top-down *ou* bottom-up

Une fois que l'on a identifié une relation de récurrence se prêtant bien à une solution par programmation dynamique, on a deux choix :

- l'approche de bas en haut (*bottom-up*) consiste (en gros) à résoudre d'abord toutes les instances de taille 1, puis celles de taille 2 et ainsi de suite jusqu'à celle qui nous intéresse. Typiquement, on obtient une solution itérative ;
- l'approche de haut en bas (*top-down*) consiste à traduire directement la relation de récurrence (en une fonction récursive, donc) et à mémoïser les appels.

En règle générale, l'approche *top-down* est plus simple à écrire (et donc à privilégier, sauf si l'on s'interdit la récursivité). Au niveau des performances (et surtout de la complexité en espace), tout dépend du problème. S'il est simple de déterminer exactement quels calculs vont être nécessaires, l'approche *bottom-up* peut être intéressante. En particulier, on peut dans certains cas ne garder en mémoire que les résultats récents (ceux correspondant à des instances de taille $n-1$), et donc faire passer la complexité en espace de $O(n^2)$ à $O(n)$ (ou de $O(n)$ à $O(1)$, ou...). En revanche, s'il n'est pas évident de déterminer *a priori* quelles instances il va réellement falloir résoudre, l'approche *top-down* peut éviter des calculs inutiles.

Exercice 6

> *Preons un exemple sans autre intérêt que sa simplicité : Fibonacci. Que donnerait, au niveau de la complexité en temps et en espace (à la louche) :*
>
> 1. *une version récursive naïve ?*
> 2. *une version récursive mémoïsée (*top-down*) ?*
> 3. *une version dynamique* bottom-up *bête ? et (semi-)intelligente ?*

2.3 Quelques problèmes très classiques

2.3.a Parenthésage optimal

- **Problème :** étand donné un produit de n matrices $A_1 \cdot \cdots \cdot A_n$ à effectuer, et en considérant par exemple que la multiplication d'un matrice (n, p) par un matrice (p, q) demande npq mutliplications élémentaires, choisir un parenthésage optimal (minimisant le nombre total de multiplications élémentaires).
- **Solution dynamique :** on calcule le parenthésage optimal pour le calcul de $\prod_{k=i}^{j} A_k$, et ce pour $1 \leqslant i \leqslant j \leqslant n$.
- **Top-down ou bottom-up :** peu importe.

2.3.b Plus longue sous-séquence croissante

- **Problème :** on nous donne une suite finie $a_1, \ldots, a_n$ dont les éléments sont ordonnables, et l'on veut la plus longue sous-séquence (*i.e.* suite extraite) $a_{\varphi(1)}, \ldots, a_{\varphi(k)}$ avec $\varphi(1) < \cdots < \varphi(k)$ et $a_{\varphi(1)} \leqslant \cdots \leqslant a_{\varphi(k)}$.
- **Solution dynamique :** pour chaque i de $[\![1, n]\!]$, on détermine la longueur de la plus longue sous-séquence croissante dont le dernier élément est a_i.
- **Top-down ou bottom-up :** peu importe.
- **Remarque :** la solution dynamique est en $O(n^2)$, il y a aussi une (superbe) solution en $O(n \ln n)$ à base de réussite : je vous invite à chercher *patience sorting* dans Google.

2.3.c Distance d'édition

- **Problème :** étant données deux chaînes s et s', et les opérations élémentaires «suppression», «insertion» et «substitution» (d'un caractère, dans chacun des cas), déterminer le nombre minimal d'opérations pour passer de s à s' (et la suite d'opérations à effectuer).
- **Solution dynamique :** on calcule les distances entre chacun des préfixes de s et chacun des préfixes de s'.
- **Top-down ou bottom-up :** on peut faire passer la complexité en espace de $O(|s| \cdot |s'|)$ à $O(|s| + |s'|)$ avec une approche bottom-up, en réalisant que l'on peut jeter tout ce qui ne date pas de la dernière étape.

2.4 Et d'autres moins classiques, ou rentrant moins dans le cadre de départ

2.4.a Partitions d'un entier

Une partition d'un entier n est une suite finie d'entiers $\lambda_1, \dots, \lambda_k$ tels que $\lambda_1 \leqslant \dots \leqslant \lambda_k$ et $\lambda_1 + \dots + \lambda_k = n$. Par exemple, les partitoins de 5 sont :

- 1 + 1 + 1 + 1 + 1
- 1 + 1 + 1 + 2
- 1 + 1 + 3
- 1 + 2 + 2
- 1 + 4
- 2 + 3
- 5

On veut déterminer le nombre de partitions de n [6].

Un instant [7] de réflexion permet de réaliser qu'en notant $f(n,k)$ le nombre de partitions de n n'utilisant que des entiers inférieurs ou égaux à k, on a $f(n,k) = f(n,k-1) + f(n-k,k)$, ce qui donne immédiatement une solution dynamique en $O(n^2)$.

Remarques

- L'informaticien s'arrête là, rien n'empêche le matheux de commencer à parler de fonctions génératrices voire d'aller lire les milliers de pages érites sur le sujet…
- Ici, on a un problème de dénombrement et non d'optimisation. C'est une autre classe de problèmes pour laquelle la programmation dynamique est souvent efficace. On pourra s'muser à chercher le nombre de $n \leqslant 10^{20}$ dont la représentation en base 10 ne contient jamais trois chiffres consécutifs dont la somme dépasse 10 ou autres bêtises du même genre…

2.4.b Plus grand bloc équilibré

On considère un tableau t de booléens de taille n, et l'on cherche le plus grand bloc de la forme `t[i : j]` contenant autant de `True` que de `False`.

On sort un peu ici de la programmation dynamique, mais certaines idées retent les mêmes : on va calculer pour chaque i de $[\![1, n]\!]$ la différence d_i entre le nombre de `True` et le nombre de `False` dans `t[:i]`. Un bloc `t[i : j]` sera équilibré ssi $d_i = d_j$. L'astuce consiste à créer un tableau indicé non pas par les i mais par les valeurs possibles de d_i. Dans la case k, on stockera 0 si l'on n'a jamais rencontré la valeur k pour un d_i. Si la valeur k a été rencontrée, on stockera le couple (p, q), où p est le plus petit indice tel que $d_p = k$ et q le plus grand. Je vous laisse vous convaincre que :

- il n'est pas très difficile de remplir ce tableau en une seule passe sur le tableau de départ (et donc en $O(n)$) ;
- une fois ce tableau construit, il est immédiat de récupérer le plus grand bloc équilibré en $O(n)$.

Cependant, une fois qu'on a compris l'algortihme et qu'on essaie de l'implémenter, on se heurte à une petite difficulté : puisque le tableau auxiliaire est naturellement indicé par les valeurs des différences, on a envie que ses indices aillent de $-n$ à n. Évidemment, il n'est pas trsè difficile de faire les décalages adéquats, mais le risque d'erreur augmente. Passons pour l'instant, et regardons ce qui se passe dans un cas un peu plus ciompliqué.

On a maintenant trois valeurs possibles pour chaque case : 0, 1 et 2. On veut toujours le plus grand bloc équilibré, sauf que cela signifie maintenant «qui contient autant de 0 que de 1 que de 2». Vu qu'on avait trouvé une méthode marchant bien, on essaie de l'adapter : on calcule les différences d_i (nombre de 1 moins nombre de 0) et $d'i$ (nombre de 2 moins nombre de 1), on remarque que `t[i : j]` est équilibré ssi $(d_i, d'_i) = (d_j, d'_j)$. Arrivés là, on se dit que l'on voudrait un tableau indicé par les valeurs du couple $(d_i, d'i)$: on est très fort en maths, on sait qu'il suffit de prendre une matrice $(2n+1) \times (2n+1)$. Sauf que…initialiser une telel matrice ne se fait pas en $O(n)$!

6. le lecteur perspicace remarquera que cela revient à déterminer le nombre de manières «vraiment distinctes» de découper une barre de toblerone…

7. notez bien qu'un instant, ça peut être long !

C'est un peu dommage, parce qu'il est clair qu'on ne va remplir que n cases au maximum. On peut ruser[8], mais le plus simple est de choisir (enfin !) une structure de données appropriée : un dictionnaire.

Pour ce que l'on va en faire ici, un dictionnaire est un tableau dans lequel les «indices» (que l'on appellera *clés*) peuvent prendre des valeurs arbitraires[9]. Si d est un dictionnaire, on peut, en temps constant :

- associer la valeur v à la clé k par `d[k] = v` ;
- déterminer s'il y a déjà une valeur associée à une certaine clé k par `k in d` (qui renvoie un booléen) ;
- récupérer la valeur associée à une clé k par `d[k]` (s'il n'y a pas de valeur associée à k, on aura bien sûr une erreur).

On peut de plus itérer sur toutes les clés de d par `for k in d:` et sur toutes les valeurs de d par `for v in d.values():`.

Avec cet outil, il devient possible d'écrire une fonction résolvant le problème à 3 valeurs en temps linéaire (et il devient accessoirement plus facile d'écrire une fonction résolvant le problème à 2 valeurs).

Exercice 7

> **1.** *Écrire une fonction résolvant le problème avec deux valeurs en utilisant un tableau auxiliaire.*
> **2.** *Écrire une fonction résolvant le problème avec deux valeurs en utilisant un dictionnaire auxiliaire. Comparer le nombre d'erreurs d'indices que vous avez eu à corriger avant que ça marche…*
> **3.** *Écrire une fonction résolvant le problème avec trois valeurs en utilisant un dictionnaire auxiliaire.*

2.4.c Autour de Syracuse

Pour tout entier $p \in \mathbb{N}^*$, on définit la suite de Syracuse associée par

$$u_0 = p \text{ et } \forall n \in \mathbb{N},\ u_{n+1} = \begin{cases} \frac{u_n}{2} & \text{Si } u_n \text{ pair} \\ 3u_n + 1 & \text{Sinon} \end{cases}$$

On admet que pour toute valeur raisonnable de p[10], la suite finit par boucler sur $1, 4, 2$. On note $syr(p)$ le plus petit n tel que $u_n = 1$.

Exercice 8

> *Écrire une fonction récursive (toute simple)* `syr(p)`.

Si l'on veut calculer *une* valeur de $syr(p)$, on n'a guère d'autre choix que de procéder bêtement. Dès que l'on calcule plusieurs valeurs, en revanche, on a toutes les chances de retomber lors d'un calcul intermédiaire sur une valeur déjà calculée : par exemple, en calculant $syr(3)$, on suit la chaîne $3 \to 10 \to 5 \to 16 \to 8 \to 4 \to 2 \to 1$. Si l'on veut un peu plus tard calculer $syr(5)$, il serait judicieux de ne pas tout recommencer.

Supposons par exemple que l'on souhaite déterminer le maximum des $syr(k)$ pour $1 \leqslant k \leqslant n$. Vu le comportement complexe de la suite, il est impossible de prévoir à l'avance quelles valeurs on va être amenés à calculer : impossible ici de procéder en *bottom-up*. Pour la même raison, le cache sera nécessairement un dictionnaire : on ne peut pas borner les k pour lesquels on va calculer $syr(k)$, et même si on le pouvait cette borne serait beaucoup trop grande. Morale de l'histoire : une fonction récursive mémoïsée utilisant un dictionnaire. On obtient alors ce genre d'arbres d'appels :

8. je vous laisse chercher…
9. pas forcément entières, même si ce sera le cas ici
10. Aux dernières nouvelles, cela a été vérifié jusqu'à $p = 5 \times 2^{60} \simeq 5.8 \times 10^{18}$.

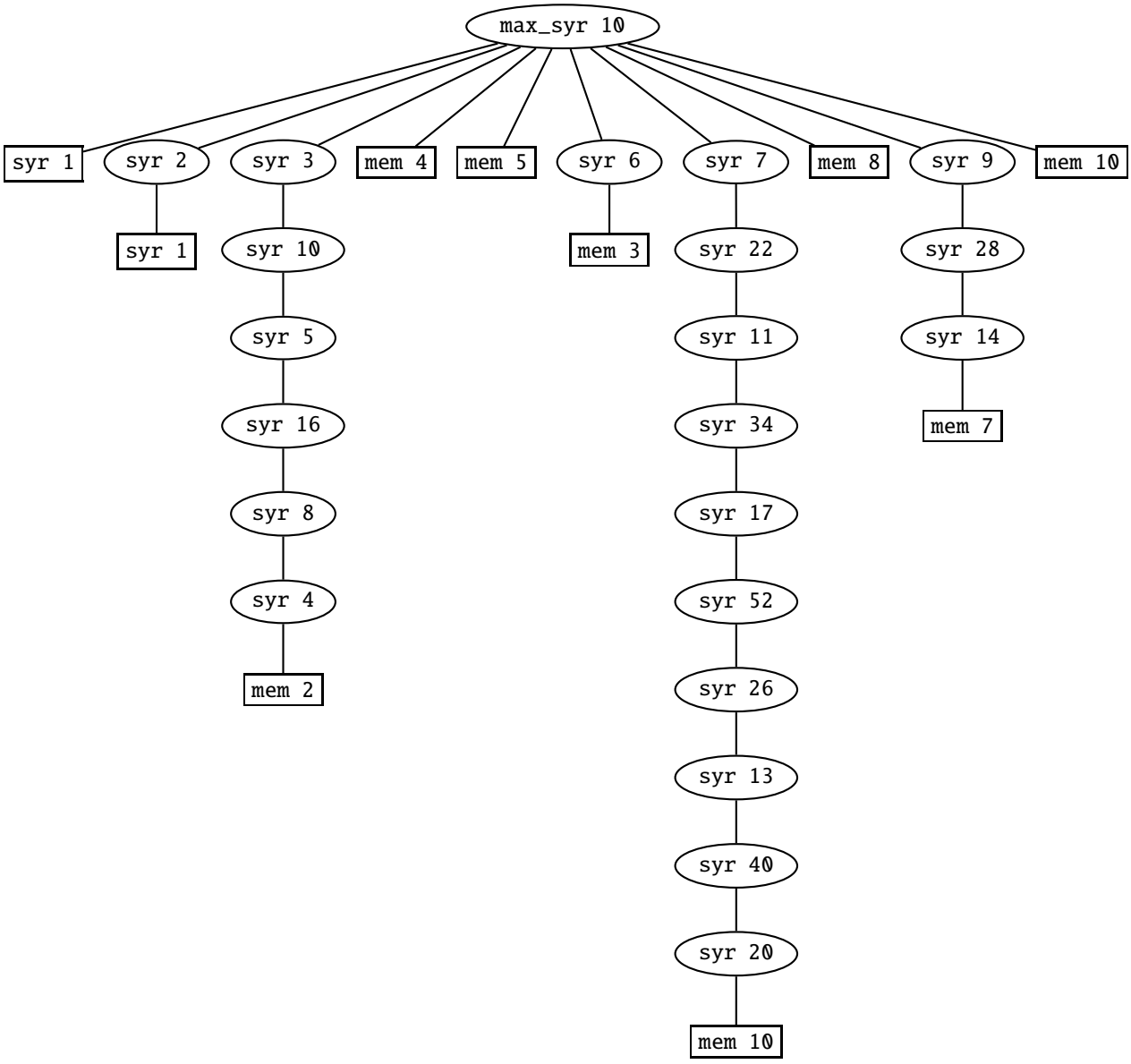

Figure 4 – Maximum des temps de convergence pour Syracuse, de 1 à 10, mémoïsé.

Exercice 9

1. *Quelle est la plus grande valeur de* $f(p)$ *pour* $1 \leqslant p \leqslant 1000$ *? Et jusqu'à* $p = 10^7$ *?*
2. *Combien d'entiers* $p \leqslant 10^6$ *vérifient-ils* $f(p) \leqslant 30$ *?*
3. *Combien d'entiers* $p \in \mathbb{N}^*$ *vérifient-ils* $f(p) \leqslant 50$ [a] *?*

a. il va falloir choisir une autre approche…