CONCEPTION ALGORITHMIQUE

OLIVIER BODINI

Résumé. Nous abordons, dans cette séance, les algorithmes de type Programmation Dynamique. Nous illustrons ce type de programmation par un premier exemple introductif concernant le calcul du n-ième nombre de Fibonacci, puis par un algorithme plus conséquent permettant de construire des arbres binaires de recherche optimaux.

1. Programmation Dynamique

La programmation dynamique, comme le principe Diviser pour Régner, est un principe algorithmique reposant sur une approche récursive des problèmes. Certains problèmes, quand on les scinde, font appel plusieurs fois aux mêmes sous-problèmes, ou à des sous-problèmes qui ne sont pas indépendants les uns des autres. Dans ce cas, les méthodes Diviser pour Régner ne sont pas aussi efficaces. En voici un exemple très simple. La suite de Fibonacci est définie récursivement par $F_0 = 0$, $F_1 = 1$ et $F_{n+2} = F_{n+1} + F_n$ pour $n \geq 0$. Supposons que l'on cherche à calculer le k-ième terme de cette suite. Si on utilise directement le principe Diviser pour Régner suivant :
Diviser : On remarque que pour calculer F_k, il suffit de savoir calculer F_{k-1} et F_{k-2}.
Régner : On résout le problème récursivement pour F_{k-1} et F_{k-2} si l'indice est supérieur à 2, sinon on remplace directement.
Combiner : On additionne F_{k-1} et F_{k-2}.

On obtient le pseudo-code ci-dessous :

Algorithme 1 : FIBO

Entrées : Un entier positif n.
Sorties : Le n-ième nombre de Fibonacci.

```
1 FIBO(n)
2 si n < 2 alors
3 |  retourner n
4 sinon
5 |  retourner FIBO(n − 1)+FIBO(n − 2)
```

Analyse de l'algorithme FIBO :
Preuve de Terminaison :
Immédiat.
Preuve de Validité :
Immédiat.
Analyse de la Complexité en nombre d'additions :
On a clairement la relation de récurrence suivante pour le nombre d'additions : $\mathcal{T}(0) = 0$, $\mathcal{T}(1) = 0$ et $\mathcal{T}(n) = \mathcal{T}(n-1) + \mathcal{T}(n-2) + 1$. Un calcul rapide montre que $\mathcal{T}(n) = \Theta(\phi^n)$ avec $\phi = (1+\sqrt{5})/2$.

Cette approche se révèle trop couteuse, car on calcule plusieurs fois les mêmes sous-problèmes comme le montre le déroulement de l'algorithme pour trouver F_5 :
1er niveau de récursion : Pour trouver F_5, on calcule F_4 et F_3
2ème niveau : Pour trouver F_4, on calcule F_3 et F_2 et pour trouver F_3, on calcule F_2.
3ème niveau : Pour trouver F_3, on calcule F_2

En conclusion, pour trouver F_5, on a calculé 1 fois F_4, 2 fois F_3, 3 fois F_2. Un algorithme Diviser pour Régner fait plus de travail que nécessaire, en résolvant plusieurs fois des sous-problèmes identiques. Alors qu'évidement, il aurait suffit de les calculer chacun une fois ! La programmation dynamique permet de pallier à ce problème. Pour ce faire, un algorithme de type Programmation Dynamique résout chaque sous-problème une seule fois et mémorise sa réponse dans un tableau, évitant ainsi le recalcul de la solution chaque fois que le sous-problème est rencontré de nouveau.

Voici le pseudo-code d'un algorithme de type Programmation Dynamique pour trouver la valeur du n-ième nombre de Fibonacci. Attention, il faut faire l'initialisation du tableau en dehors de la fonction récursive.

Algorithme 2 : FIGO-PG-REC

Entrées : Un entier positif n.
Sorties : Le n-ième nombre de Fibonacci.

```
1  INITIALISATION(n)
2  Créer un tableau T de longueur n initialisé à Nil;
3  T[0] :=0 ; T[1] :=1;

4  FIBO-PG-REC(n)
5  si n < 2 alors
6      retourner n
7  si T[n − 1] = Nil alors
8      T[n − 1] := FIBO-PG-REC(n − 1) (on verra pourquoi il est inutile de tester T[n − 2] = Nil)
9  T[n] := T[n − 1] + T[n − 2];
10 retourner T[n]
```

Analyse de l'algorithme FIBO-PG-REC :

Preuve de Terminaison :
FIBO-PG-REC(n) s'arrête quand n vaut 0 ou 1. Comme FIBO-PG-REC(n) appelle uniquement (de manière directe) FIBO-PG-REC($n-1$). On en déduit que si FIBO-PG-REC($n-1$) s'arrête, alors l'algorithme FIBO-PG-REC(n) s'arrête aussi. Par induction, on a la preuve de terminaison.

Preuve de Validité :
Par induction, montrons qu'à la sortie de FIBO-PG-REC(n), pour tout $0 \leq i \leq n$, on a $T[i] = F_i$ et pour $j > n$, on a $T[j]$ =Nil. C'est vrai quand n vaut 1. Supposons qu'à la sortie de FIBO-PG-REC(n) T soit bien rempli jusqu'à la case n et pour $j > n$, on a $T[j]$ =Nil, alors quand on lance FIBO-PG-REC($n+1$), à la ligne 8, on fait l'appel FIBO-PG-REC(n) et donc le tableau T est alors rempli jusqu'à n, enfin ligne 9, on remplit correctement la case $n+1$. Donc à la sortie de FIBO-PG-REC($n+1$) le tableau est bien rempli jusqu'à $n+1$ et vide après. Par récurrence, la validité de l'algorithme s'ensuit.

Analyse de la Complexité en nombre d'additions :
FIBO-PG-REC($n+1$) fait une addition ligne 9 et appelle FIBO-PG-REC(n) ligne 8. Donc le nombre d'additions vérifie : $\mathcal{T}(0) = 0$, $\mathcal{T}(1) = 0$ et $\mathcal{T}(n+1) = \mathcal{T}(n) + 1$. Ceci donne $\mathcal{T}(n) = \Theta(n)$.

On privilégiera donc cette approche quand la solution d'un problème sur une instance de taille n s'exprime en fonction de solutions du même problème sur des instances de taille inférieure à n et qu'une implémentation récursive du type diviser pour Régner conduit à rappeler de nombreuses fois les mêmes sous-problèmes.

Il y a en fait deux possibilités pour implémenter un algorithme de type Programmation Dynamique qui dépendent de la manière où l'on remplit le tableau (itérative ou récursive) :

– Une version récursive, appelée aussi *memoizing*, pour laquelle à chaque appel, on regarde dans le tableau si la valeur a déjà été calculée . Si c'est le cas, on récupère la valeur mémorisée. sinon, on la calcule et on la stocke. Cette méthode permet de ne pas avoir à connaitre à l'avance les valeurs à calculer. C'est celle utilisée pour le pseudo-code de FIBO-PG-REC.

– Une version itérative où l'on initialise les cases correspondant aux cas de base. Puis on remplit le tableau selon un ordre permettant à chaque nouveau calcul de n'utiliser que les solutions déjà calculées.

Cette dernière donne pour le calcul du n-ième nombre de Fibonacci, le pseudo-code suivant :

Algorithme 3 : FIGO-PG-IT

Entrées : Un entier positif n.
Sorties : Le n-ième nombre de Fibonacci.

```
1 FIBO-PG-IT(n)
2 Créer un tableau T de longueur n ; T[0] :=0 ; T[1] :=1;
3 pour i allant de 2 à n faire
4 |  T[i] := T[i − 1] + T[i − 2]
5 retourner T[n]
```

On remarque que dans ce cas très simple, il est même possible de ne pas créer de tableau en faisant :

Algorithme 4 : FIGO-PG-IT2

Entrées : Un entier positif n.
Sorties : Le n-ième nombre de Fibonacci.

```
1 FIBO-PG-IT2(n)
2 si n < 2 alors
3 |  retourner n
4 i := 0 ; j := 1;
5 pour k allant de 2 à n faire
6 |  temp := j ; j := j + i ; i := temp;
7 retourner j
```

Analyse de l'algorithme FIBO-PG-IT2 :
Preuve de Terminaison :
Immédiat.
Preuve de Validité :
Considérons l'invariant de boucle sur k suivant : i vaut F_{k-2} et j vaut F_{k-1}. Quand on entre pour la première fois dans la boucle c'est vrai. Maintenant, supposons qu'au début de la k-ième itération, on a $i = F_{k-2}$ et $j = F_{k-1}$, alors, ligne 4, j devient F_k et i prend la valeur de *temp* qui est $j = F_{k-1}$. Donc l'invariance est préservée. A la sortie de la boucle j vaut donc F_n, ce qui est le résultat attendu.
Analyse de la Complexité en nombre d'additions :
FIBO-PG-IT2(n) fait une addition à chaque itération de la boucle sur k, soit $n-1$ additions. Ceci donne donc $\mathcal{T}(n) = \Theta(n)$.

1.1. Arbres, Arbres binaires, Arbres binaires de recherche optimaux.

1.1.1. *Arbres.* Passons un instant sur la définition d'arbres. Nous introduisons ici des arbres enracinés planaires (c'est à dire ayant un sommet distingué (la racine) et tel que les fils d'un noeud formé une liste ordonnée. la notion d'arbres est omniprésente dans les sciences (arbres phylogénétiques en biologie, arbres des choix et processus à branchement en probabilité, arbres des appels récursifs, arbres d'expressions et arbres de recherche en informatiques), car elle permet de modéliser les structures hiérarchiques simples. Pour être précis, nous commençons par introduire la notion de classe combinatoire.

Définition 1. *Une* classe combinatoire $\mathcal{C} = (E, t)$ *est constitué d'un (multi-)ensemble* E *et d'une fonction* t *de* E *dans* $\mathbb{N}$ *tel que pour tout* $k > 0, \{a \in E, t(a) = k\}$ *est fini. Les éléments de* E *sont appelés des* objets *et pour tout objet* a, $t(a)$ *est appelé la* taille *de* a.

Nous allons construire récursivement des classes combinatoires, pour se faire, il nous faut définir des classes de bases et des constructeurs.

La classe $\mathcal{E} = (\{\epsilon\}, \epsilon \mapsto 0)$ est appelée la classe *neutre*, elle contient un unique élément de taille 0. La classe $\mathcal{Z} = (\{\square\}, \square \mapsto 1)$ est appelée la classe *atomique*, elle contient un unique élément de taille 1.

Soit $\mathcal{C}_1 = (E_1, t_1)$ et $\mathcal{C}_2 = (E_2, t_2)$, on définit l'*union disjointe*, noté $\mathcal{C}_1 + \mathcal{C}_2$, comme la classe $(E_1 \cup E_2, t)$ où $t(a)$ vaut $t_i(a)$ si $a \in E_i$. Noter que si $a \in E_1 \cap E_2$ alors a est présent dans $E_1 \oplus E_2$ avec une multiplicité qui est la somme de sa multiplicité dans E_1 et dans E_2.

Soit $\mathcal{C}_1 = (E_1, t_1)$ et $\mathcal{C}_2 = (E_2, t_2)$, on définit le *produit*, noté $\mathcal{C}_1 \times \mathcal{C}_2$, comme la classe $(E_1 \times E_2, t)$ où $t((a, b))$ vaut $t_1(a) + t_2(b)$.

Définition 2. *Un* arbre binaire *est soit vide, soit formé d'un noeud, sa racine, et de deux sous-arbres binaires, l'un appelé* fils gauche*, l'autre* fils droit*.*

Propriété 1. *La classe* $\mathcal{T}$ *des arbres binaires dont la taille est le nombre de noeuds internes vérifie l'équation structurelle récursive* $\mathcal{T} = \mathcal{E} + \mathcal{T} \times \mathcal{Z} \times \mathcal{T}$.

On parle parfois d'arbre binaire *planaire* pour mentionner que l'on distingue la gauche de la droite.

Nous nous intéressons aux arbres permettant d'accéder à des informations. Pour cela, chaque noeud contient une *clé* qui est généralement un nombre. Un arbre non vide est donc entièrement décrit par le triplet (fils gauche, clé de la racine, fils droit). Cette définition récursive se traduit directement en une spécification de programmation.

La définition récursive des arbres binaires (comme pour les listes) conduit naturellement à des algorithmes de manipulations qui seront récursifs (voir TDs, pour des algorithmes calculant la taille, la hauteur, le nombre de feuilles,...).

Nous pouvons par exemple définir 3 types de parcours des noeuds d'un arbre binaire.

Le parcours préfixe :

Algorithme 5 : Préfixe

Entrées : Un arbre binaire T.

Sorties : Rien mais affiche le parcours préfixe de T.

```
1 Préfixe(T)
2 si NonVide(T) alors
3 |  Afficher la clé de la racine(T);
4 |  Préfixe(ArbreGauche(T));
5 |  Préfixe(ArbreDroit(T));
```

Le parcours suffixe ou postfixe :

Algorithme 6 : Suffixe

Entrées : Un arbre binaire T.
Sorties : Rien mais affiche le parcours suffixe de T.

```
1 Suffixe(T)
2 si NonVide(T) alors
3 |   Suffixe(ArbreGauche(T));
4 |   Suffixe(ArbreDroit(T));
5 |   Afficher la clé de la racine(T);
```

Le parcours infixe :

Algorithme 7 : Infixe

Entrées : Un arbre binaire T.
Sorties : Rien mais affiche le parcours infixe de T.

```
1 Infixe(T)
2 si NonVide(T) alors
3 |   Infixe(ArbreGauche(T));
4 |   Afficher la clé de la racine(T);
5 |   Infixe(ArbreDroit(T));
```

Définition 3. *Un arbre général (planaire) est un arbre dont chaque noeud a un nombre quelconque de fils ordonné de gauche à droite.*

Dans point de vue informatique, il semble donc naturel de représenter les fils d'un noeud dans une liste (chaînée). Ainsi, chaque noeud contient deux références, celle à son fils aîné (le plus à gauche des fils), et celle à son frère cadet (celui qui se trouve juste à sa droite).

Mais cette représentation donne explicitement une correspondance entre un arbre général et un arbre binaire. Il suffit de rebaptiser le fils ainé, fils gauche et le frère cadet, fils droit.

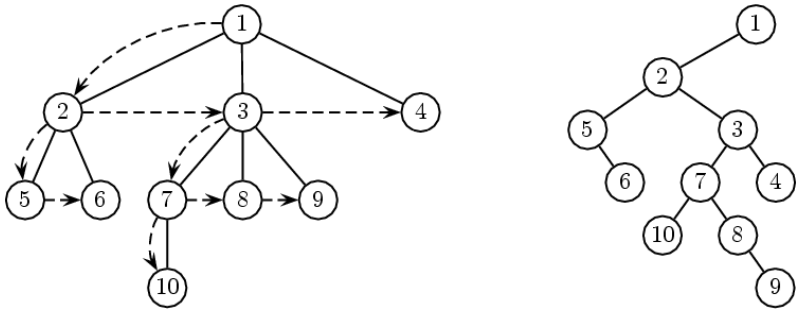

Figure 1. Correspondance arbres généraux - arbres binaires

Définition 4. *Un arbre binaire T est un* arbre binaire de recherche *si, pour tout noeud s de T, les contenus des noeuds du sous-arbre gauche de s sont strictement inférieurs au contenu de s, et que les contenus des noeuds du sous-arbre droit de s sont strictement supérieurs au contenu de s.*

Propriété 2. *Dans un arbre binaire de recherche, le parcours infixe fournit les contenus des noeuds en ordre croissant.*

Propriété 3. *Si un noeud possède un fils gauche, son prédécesseur dans le parcours infixe est le noeud le plus à droite dans son sous-arbre gauche. Ce noeud n'est pas nécessairement une feuille, mais il n'a pas de fils droit.*

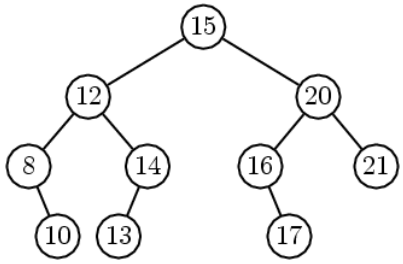

FIGURE 2. Un arbre binaire de recherche

1.1.2. *Opérations élémentaires sur les arbres binaires de recherche.* Nous allons proposer des algorithmes pour la recherche, l'insertion et la suppression d'une clef dans un arbre binaire de recherche.

Algorithme 8 : Recherche

Entrées : Un arbre binaire T, un entier x.
Sorties : .

```
1 Recherche(T,x)
2 si Vide(T) alors
3 |  retourner Faux;
4 si x=racine(T) alors
5 |  retourner Vrai;
6 si x <racine(T) alors
7 |  Recherche(ArbreGauche(T,x));
8 sinon
9 |  Recherche(ArbreDroit(T,x));
```

Algorithme 9 : Insérer

Entrées : Un arbre binaire T, un entier x.
Sorties : T avec x inséré.

```
1 Insérer(T,x)
2 si Vide(T) alors
3 |  retourner CréerNoeud(Null,x,Null);
4 si x <racine(T) alors
5 |  ArbreGauche(T)=Inserer(x, ArbreGauche(T));
6 sinon
7 |  ArbreDroit(T)=Inserer(x, ArbreDroit(T));
```

La suppression d'une clé dans un arbre est une opération plus complexe. Elle s'accompagne de la suppression d'un noeud. Comme on le verra, ce n'est pas toujours le noeud qui porte la clé à supprimer qui sera enlevé. Soit s le noeud qui porte la clé x à supprimer. Trois cas sont à considérer selon le nombre de fils du noeud x :

- si le noeud s est une feuille, alors on l'élimine ;
- si le noeud s possède un seul fils, on élimine s et on « remonte » ce fils.
- si le noeud s possède deux fils, on cherche le prédécesseur t de s. On remplace la clé de s par la clé de t, et on élimine t.

Nous allons décomposer l'algorithme de suppression en trois parties. La première, Supprimer, recherche le noeud portant la clé à supprimer et appel la deuxième, SupprimerRacine. Elle effectue la suppression selon les cas énumérés ci-dessus. La troisième, dernierDescendant est une méthode auxiliaire qui calcule le prédécesseur d'un n ?ud qui a un fils gauche.

Algorithme 10 : Supprimer

Entrées : Un arbre binaire T, un entier x.
Sorties : T avec x supprimé.

```
1 si Vide(T) alors
2 |  retourner T;
3 si x est la clé de la racine de T alors
4 |  retourner SupprimerRacine(T);
5 si x <racine(T) alors
6 |  ArbreGauche(T)=Supprimer(x, ArbreGauche(T));
7 sinon
8 |  ArbreDroit(T)=Supprimer(x, ArbreDroit(T));
9 retourner T
```

Algorithme 11 : SupprimerRacine

Entrées : Un arbre binaire T
Sorties : T sans sa racine.

```
1 si ArbreGauche(T) est vide alors
2 |  retourner ArbreDroit(T);
3 si ArbreDroit(T) est vide alors
4 |  retourner ArbreGauche(T);
5 dD = dernierDescendant(ArbreGauche(T));
6 La clé de la racine de T := la clé de la racine de dD;
7 ArbreGauche(T) := Supprimer(clé de la racine de dD, ArbreGauche(T)) ;
```

Et enfin,

Algorithme 12 : dernierDescendant

Entrées : Un arbre binaire T
Sorties : T sans sa racine.

```
1 si ArbreDroit(T) est vide alors
2 |  retourner T;
3 retourner dernierDescendant(ArbreDroit(T));
```

La récursivité croisée entre les algorithmes Supprimer et SupprimerRacine peut dérouter au premier abord. En fait, l'appel à Supprimer à la dernière ligne de SupprimerRacine conduit au noeud prédécesseur de la racine de l'arbre, appelé dD. Comme ce noeud n'a pas deux fils, il n'appelle pas une deuxième fois la méthode SupprimerRacine.

1.1.3. *Arbre de recherche optimal.* Supposons que l'on dispose d'un arbre de recherche dans lequel on fait de multiples accès (multiples recherches), comme par exemple un arbre binaire stockant les mots d'un dictionnaire en ligne où des internautes iraient chercher des définitions. Imaginons, maintenant que les recherches effectuées ne sont pas équiprobables, mais que pour chaque clé k_i (dans notre exemple, les mots), on a la probabilité p_i qu'une recherche concerne la clé k_i (par exemple, on sait que les recherches faites sur le mot *stalactite* sont plus fréquentes que celles sur le mot *lithopédion*). Comment doit-être construit cet arbre de sorte que le coût moyen de recherche soit optimisé ? Pour répondre à cette question, il nous faut au préalable définir la notion de coût. Nous considérons ici que le coût d'une recherche dans un arbre binaire est le nombre de noeuds testés pour trouver la clé. On a donc que le coût de recherche d'une clé k_i dans un arbre binaire A est $h_A(k_i) + 1$ (où la hauteur h de k_i dans A est le nombre d'arêtes de la chaîne qui va de k_i à la racine). De sorte que le coût moyen de recherche dans A est :

$$m_A = \sum_{i=1}^{n} (h_A(k_i) + 1)\, p_i$$

Le problème peut donc être posé ainsi : Etant données une séquence $K = (k_1, k_2, ..., k_n)$ de n clés distinctes triées en ordre croissant et $P = (p_1, ..., p_n)$ la suite des probabilités associées à chaque clé. On cherche un arbre binaire de recherche A pour stocker K qui minimise m_A.

1.1.4. *Structure d'un arbre binaire de recherche optimal.*

Lemme 1. *Soient A un arbre binaire, k_t une clé et A_{k_t} le sous-arbre binaire de A de racine k_t composé de tous les descendants de k_t, alors il existe i et j tels que $1 \leq i \leq j \leq n$ et A_{k_t} soit l'arbre induit par les clés $k_i, ..., k_j$.*

Preuve. Par induction sur la hauteur de k_i dans A. Si $h_A(k_i) = 0$ alors k_i est la racine de A et $A_{k_i} = A$. Supposons le lemme démontré pour les sous-arbres A_{k_i} avec $h_A(k_i) = n - 1$, soit A_{k_j} avec $h_A(k_j) = n$. Alors notons k_t le père de k_j dans A. Par induction, les noeuds de A_{k_t} sont $k_{i'}, ..., k_{j'}$ pour un certain couple (i', j'). Si k_j est le fils gauche (resp. droit) de k_t alors les noeuds de A_{k_j} sont $k_{i'}, ..., k_{t-1}$ (resp. $k_{t+1}, ..., k_{j'}$). Le lemme s'ensuit. □

Lemme 2 (propriété de la sous-structure optimale.)**.** *Supposons que A soit un arbre binaire de recherche optimal et A' un sous-arbre de racine k_r contenant les clés $(k_i, ..., k_j)$, alors ce sous-arbre A' est optimal pour le problème sur les clés $k_i, ..., k_j$ et les probabilités associées* $\left(\frac{p_i}{\sum_{k=i}^{j} p_k}, ..., \frac{p_j}{\sum_{k=i}^{j} p_k} \right)$ *(On normalise pour que la somme des probabilités donne 1).*

Preuve. [la technique du copier-coller] En effet, s'il y avait un sous-arbre A'' contenant les clés $k_i, ..., k_j$ dont le coût moyen est inférieur à celui de A', alors on pourrait remplacer A' dans A par A'', ce qui donnerait un arbre binaire de recherche dont le coût moyen est inférieur à celui de A. Cela contredirait l'optimalité de A. □

Nous allons utiliser cette propriété pour construire une solution optimale du problème à partir de solutions optimales de sous-problèmes. Soient les clés $k_i, ..., k_j$; l'une de ces clés, par exemple k_r $(i \leq r \leq j)$, est la racine d'un sous-arbre optimal contenant ces clés. Le sous-arbre gauche de la racine k_r contiendra les clés $k_i, ..., k_{r-1}$; le sous-arbre droit contiendra les clés $k_{r+1}, ..., k_j$. Si l'on examine tous les candidats k_r pour la place de racine (avec $i \leq r \leq j$) et si l'on détermine, pour chaque r, un arbre binaire de recherche optimal contenant $k_i, ..., k_{r-1}$ et un contenant $k_{r+1}, ..., k_j$, alors on est certain de trouver un arbre binaire de recherche optimal pour $k_i, ..., k_j$.

1.1.5. *Une formule de récurrence.* Nous pouvons dorénavant définir récursivement le coût d'une solution optimale.

Lemme 3. *Soit B un arbre binaire de recherche optimal contenant les clés $k_i, ..., k_j$ et les probabilités associées* $\left(\frac{p_i}{\sum_{k=i}^{j} p_k}, ..., \frac{p_j}{\sum_{k=i}^{j} p_k} \right)$*, on note* $e[i,j] = \sum_{t=i}^{j} (h_B(k_t) + 1) p_t$

et $w(i,j) = \sum_{k=i}^{j} p_k$. Si k_r est la racine de B, on a $e[i,j] = e[i, r-1] + e[r+1, j] + w(i,j)$.

Preuve. En effet, $e[i,j] = \sum_{t=i}^{r-1} (h_B(k_t) + 1) p_t + p_r + \sum_{t=r+1}^{j} (h_B(k_t) + 1) p_t$. Or le sous-arbre gauche B_G et le sous-arbre droit B_D sont optimaux d'après le lemme de la sous-structure optimale. Donc $\sum_{t=i}^{r-1} (h_B(k_t) + 1) p_t = \sum_{t=i}^{r-1} (h_{B_G}(k_t) + 2) p_t = \sum_{t=i}^{r-1} (h_{B_G}(k_t) + 1) p_t + w(i, r-1) = e[i, r-1] + w(i, r-1)$

et pour la même raison $\sum_{t=r+1}^{j} (h_B(k_t)+1)p_t = e[r+1,j] + w(r+1,j)$. Comme $w(i,r-1) + p_r + w(r+1,j) = w(i,j)$, on a bien la formule enoncée. □

On remarque que $\dfrac{e[i,j]}{\sum_{k=i}^{j} p_k}$ est exactement le coût moyen d'une recherche dans un arbre optimal contenant les clefs $k_i, ..., k_j$.

Cette équation récursive suppose que nous sachions quel est le noeud k_r à prendre comme racine. Or, on choisit la racine qui donne le coût de recherche moyen le plus faible ; d'où la formulation récursive finale :

$$e[i,j] = \begin{cases} 0 & \text{si } j < i \\ p_i & \text{si } j = i \\ \min\limits_{i \le r \le j} \{e[i,r-1] + e[r+1,j] + w(i,j)\} & \text{si } i < j \end{cases}$$

Les valeurs $e[i,j]$ ne donnent à un facteur normalisant près que les coûts de recherche moyens dans des arbres binaires de recherche optimaux. Elles contiennent trop peu d'informations pour permettre de reconstruire l'arbre binaire de recherche optimal. Pour y parvenir, il nous faut stocker les racines des différents sous-arbres optimaux. Pour cela, nous définissons $racine[i,j]$, pour $1 \le i \le j \le n$, comme étant l'indice r pour lequel k_r est la racine d'un arbre binaire de recherche optimal contenant les clés $k_i, ..., k_j$. Nous verrons comment calculer les valeurs de $racine[i,j]$ en même temps que $e[i,j]$.

1.1.6. *Calcul du coût de recherche moyen dans un arbre binaire de recherche optimal.* Nous allons stocker les valeurs $e[i,j]$ dans un tableau $e[1..(n+1), 0..n]$. On n'utilisera en fait que les éléments $e[i,j]$ pour lesquels $j \ge i-1$. On emploiera aussi un tableau $racine[i,j]$, pour mémoriser la racine du sous-arbre contenant les clés $k_i, ..., k_j$. De même, il ne sera utilisé dans ce tableau que les éléments pour lesquels $1 \le i \le j \le n$. On aura besoin d'un tableau pour stocker les $w[i,j]$, à des fins d'efficacité. En fait, au lieu de calculer la valeur de $w(i,j)$ chaque fois que l'on calcule $e[i,j]$ ce qui prendrait $\Theta(j-i)$ additions, il est plus économique en temps (mais pas en mémoire !) de stocker ces valeurs dans un tableau $w[1..n, 1..n]$. Pour le cas de base, on calcule $w[i,i] = p_i$ pour $1 \le i \le n$. Pour $j \ge i$, on calcule $w[i,j] = w[i,j-1] + p_j$. On peut donc calculer les $\Theta(n^2)$ valeurs de $w[i,j]$ avec un temps $\Theta(1)$ pour chacune. Nous pouvons maintenant donner le pseudo-code ABR-OPTIMAL :

Algorithme 13 : ABR-OPTIMAL-IT

Entrées : Un tableau P contenant les probabilités $p_1, ..., p_n$ associées aux clés et la taille n.
Sorties : Rien mais on a rempli les tableaux e et $racine$.

```
1  Créer un tableau e[1..(n+1), 0..n] initialisé à ∞.
2  Créer un tableau w[1..n, 1..n] et créer un tableau racine[1..n, 1..n].
3  ABR-OPTIMAL-IT(P, n)
4  e[n, n+1] := 0;
5  pour i allant de 1 à n faire
6      e[i, i-1] := 0;
7      e[i, i] := P[i];
8      w[i, i] := P[i];
9  pour l allant de 2 à n faire
10     pour i allant de 1 à n-l+1 faire
11         j := i+l-1;
12         w[i, j] := w[i, j-1] + P[j];
13         pour r allant de i à j faire
14             t := e[i, r-1] + e[r+1, j] + w[i, j];
15             si t < e[i, j] alors
16                 e[i, j] := t;
17                 racine[i, j] := r;
```

Analyse de l'algorithme ABR-OPTIMAL-IT :

Preuve de Terminaison : Immédiat, il n'y a que des boucles **Pour**.

Preuve de Validité :
La boucle **Pour** des lignes 1-8 initialise les valeurs de $e[i, i]$ et $w[i, i]$. La boucle **Pour** des lignes 9-17 utilise ensuite les récurrences trouvées pour calculer $e[i, j]$ et $w[i, j]$ pour tout $1 \leq i < j \leq n$. Dans la première itération, quand $l = 2$, la boucle calcule $e[i, i + 1]$ et $w[i, i + 1]$ pour $i = 1, 2, ..., n - 1$. La deuxième itération, avec $l = 3$, elle calcule $e[i, i + 2]$ et $w[i, i + 2]$ pour $i = 1, 2, ..., n - 2$, etc. La boucle **Pour** la plus interne, en lignes 13-17, essaie chaque indice candidat r pour déterminer quelle est la clé k_r à utiliser comme racine d'un arbre binaire de recherche optimal contenant les clés $k_i, ..., k_j$. Cette boucle **Pour** mémorise la valeur courante de l'indice r dans $racine[i, j]$ chaque fois qu'elle trouve une clé meilleure pour servir de racine.
Analyse de la Complexité en nombre d'additions :
On a clairement la valeur suivante pour le nombre d'additions : $\sum_{l=2}^{n} \sum_{i=1}^{n-l+1} \left(2 + \sum_{r=i}^{i+l-1} 2\right)$. Un simple calcul montre que $\sum_{l=2}^{n} \sum_{i=1}^{n-l+1} \left(2 + \sum_{r=i}^{i+l-1} 2\right) = (n^3 + 5n^2 - 6n)/2$. La procédure ABR-OPTIMAL fait donc $\Theta(n^3)$ additions.

Algorithme 14 : ABR-OPTIMAL-REC

Entrées : Un tableau P contenant les probabilités $p_1, ..., p_n$ associées aux clés, les tableaux e et $racine$ et deux entiers i et j.
Sorties : On récupère $e[i,j]$ et $racine[i,j]$.

```
MEMORISATION
Créer un tableau e[1..(n+1), 0..n] initialisé à ∞.
Créer un tableau w[1..n, 1..n] et créer un tableau racine[1..n, 1..n].
e[n, n+1] := 0;
pour i allant de 1 à n faire
    e[i, i-1] := 0;
    e[i, i] := P[i];
    w[i, i] := P[i];
pour l allant de 2 à n faire
    pour i allant de 1 à n − l + 1 faire
        j := i + l − 1;
        w[i, j] := w[i, j − 1] + P[j];

ABR-OPTIMAL-REC(P, e, racine, i, j)
si e[i, j] < ∞ alors
    retourner e[i, j]
sinon
    pour r allant de i à j faire
        t := ABR-OPTIMAL-REC(P, e, racine, i, r − 1) +
        ABR-OPTIMAL-REC(P, e, racine, r + 1, j) + w[i, j];
        si t < e[i, j] alors
            e[i, j] := t;
            racine[i, j] := r;
    retourner e[i, j]
```

L'Analyse de l'algorithme ABR-OPTIMAL-REC est laissée en exercice.

On peut facilement récupérer l'arbre optimal en appelant RECUP-ARBRE($racine, k, 1, n$) :

Algorithme 15 : RECUP-ARBRE

Entrées : Le tableau $racine$ et le tableau k contenant les clés i et j.
Sorties : On récupère l'arbre optimal contenant les clés $k_i, ..., k_j$.

```
RECUP-ARBRE(racine, k, i, j)
si i > j alors
    retourner ∅
sinon
    si i = j alors
        retourner (k[i], ∅, ∅) (c'est à dire une feuille d'étiquette k[i])
    sinon
        retourner
            (k[racine[i, j]], RECUP-ARBRE(racine, k, i, racine[i, j] − 1), RECUP-ARBRE(racine, k, racine[i, j] + 1, j)).
        (C'est à dire, l'arbre dont la racine est d'étiquette k[racine[i, j]] et dont l'arbre gauche (resp. droit) est
        RECUP-ARBRE(racine, k, i, racine[i, j] − 1) (resp. RECUP-ARBRE(racine, k, racine[i, j] + 1, j)))
```

E-mail address: olivier.bodini@lipn.fr