4MMAOD6 - TD 5 : Branch & Bound

Préambule : Synthèse du cours sur Branch&Bound. Soit A un ensemble fini, potentiellement de très grande taille. On considère le problème de minimisation (un problème de maximisation se traite de manière similaire) : *Trouver* $\bar{x}$ *qui atteint* $\min_{x\in A} f(x)$.
Un algorithme de *Branch&Bound* (ou *séparation - évaluation* en français) consiste à

1. *solution courante* s : partant d'une solution initiale $s \in A$ non nécessairement optimale, mettre à jour s jusqu'à prouver que $f(s)$ est optimale ;
2. *Branch* : explorer de manière arborescente A ; un nœud de l'arbre représente donc un sous-ensemble de A, donc un sous-problème ; un sous-ensemble de cardinal 1 est une feuille. Typiquement, l'opération Branch permet de parcourir les fils d'un nœud.
3. *Bound* : élaguer les nœuds correspondant à un sous ensemble $X \subset E$ dont on peut vérifier facilement que tous les éléments sont moins bons que $f(s)$; i.e. $\forall x \in X : f(x) \geq f(s)$. Pour cela, on utilise une fonction g dite d'*évaluation optimiste* qui prend en entrée un nœud n et retourne une valeur meilleure (donc inférieure pour un problème *Min*, supérieure pour un problème *Max*) que la valeur de f en toute feuille de la sous-arborescence de racine n.
   Autrement dit, g retourne un minorant (resp. majorant) de f pour un problème Min (resp. Max).

Pour le programmer, on utilise une collection E, initialisée avec les racines de l'arborescence (il peut y en avoir une ou plusieurs selon les cas) qui stocke à tout instant l'ensemble des nœuds restant à explorer. Le programme ci-dessous résout un problème **Min** f, la fonction d'évaluation optimiste g retournant un minorant de f :

```
Soit s ∈ A une solution initiale et z̄ := f(s) sa valeur;
E := { racines de l'arbre à explorer };
répéter
    Choisir P ∈ E et le supprimer de E  E := E \ {P} ;
    Évaluer P de manière optimiste : z = g(P);
    si Si on peut montrer z = min_{s∈P} f(s) (par ex. si P ne contient qu'un élément) alors
        mettre à jour z̄ := z et s si besoin
    ;
    sinon si z < z̄ alors
        Branch Décomposer P en k sous-problèmes P_1 ⊔ ··· ⊔ P_k = P;
        E = E ∪ {P_1, ..., P_k};
jusqu'à E = ∅;
```

B&B à Utopia (30' en CTD interactif)

La ville de Utopia possède 4 districts dont les populations sont données dans le tableau qui suit. On veut constituer une assemblée représentative de 11 élus, ce qui fait un représentant pour 100 habitants.
Le problème est de définir le nombre de représentants par district de façon que l'écart maximum de représentation par personne par rapport à l'idéal de un pour cent, soit minimum.
Dans le tableau suivant #idéal représente le nombre (fractionnaire) de représentants qu'il faudrait pour satisfaire la representation idéale par électeur notée : idéal p.p.

District	population	#idéal	idéal p.p.
A	120	1,20	0,01
B	155	1,55	0,01
C	360	3,60	0,01
D	465	4,65	0,01
	———	———	
total	1100	11	

Par exemple si on affecte 1, 1, 4 et 5 représentants aux districts A, B, C et D respectivement, les écarts par rapport à l'idéal sont respectivement de $0,01 - 1/120 = 0,00167$, $0,01 - 1/155 = 0,00355$, $4/360 - 0,01 = 0,00111$ et $5/465 - 0,01 = 0,00075$. Donc l'écart maximum est de $0,00355$.

Question 1 Résoudre par *branch&bound* le problème de trouver le nombre de représentants par district minimisant l'écart maximum.

Soit x_i le nombre optimal de député dans le district i et p_i le nombre d'électeurs. Le tableau ci-dessous donne la valeur de l'écart à 1% pour le district $i \in \{A, B, C, D\}$ en fonction du nombre x_i de députés ($1 \leq x_i \leq 6$).

	p_i	$\left\|0,01 - \frac{\mathbf{1}}{p_i}\right\|$	$\left\|0,01 - \frac{\mathbf{2}}{p_i}\right\|$	$\left\|0,01 - \frac{\mathbf{3}}{p_i}\right\|$	$\left\|0,01 - \frac{\mathbf{4}}{p_i}\right\|$	$\left\|0,01 - \frac{\mathbf{5}}{p_i}\right\|$	$\left\|0,01 - \frac{\mathbf{6}}{p_i}\right\|$
A	120	0,001666667	0,006666667	0,015	0,023333333	0,031666667	0,04
B	155	0,003548387	0,002903226	0,009354839	0,015806452	0,022258065	0,028709677
C	360	0,007222222	0,004444444	0,001666667	0,001111111	0,003888889	0,006666667
D	465	0,007849462	0,005698925	0,003548387	0,001397849	0,000752688	0,002903226

La stratégie pour le Branch&Bound est définie par :

— branchement (*branch*) : choix du nœud le plus contraint ;
— élagage (*bound*) : $\left|0,01 - \frac{x_i}{p_i}\right| \geq$valeur(solution courante) ;
— stratégie de parcours : ici, meilleur d'abord
— solution initiale : ici $(x_A, x_B, x_C, x_D) = (1, 1, 4, 5)$ de valeur 0,00355.

A partir de la solution initiale s (ici $s = 0,00355$), on beut borner l'intervalle où peut se trouver x_i (cf tableau). On branche d'abord sur A, puis B, puis C et enfin D. Avec la valeur initiale (x_A, x_B, x_C, x_D)= (1,1,4,5) de valeur 0,00355 (correspondant à $X_B = 1$), les étapes sont (cf figure ci-après) :

1. $(?, ?, ?, ?)$: Branch x_A : $x_A = 1$, valeur optimiste $\geq 0,0016$;
   pour $x_A \geq 2$, valeur optimiste $> 0,0066 \Longrightarrow$ BOUND

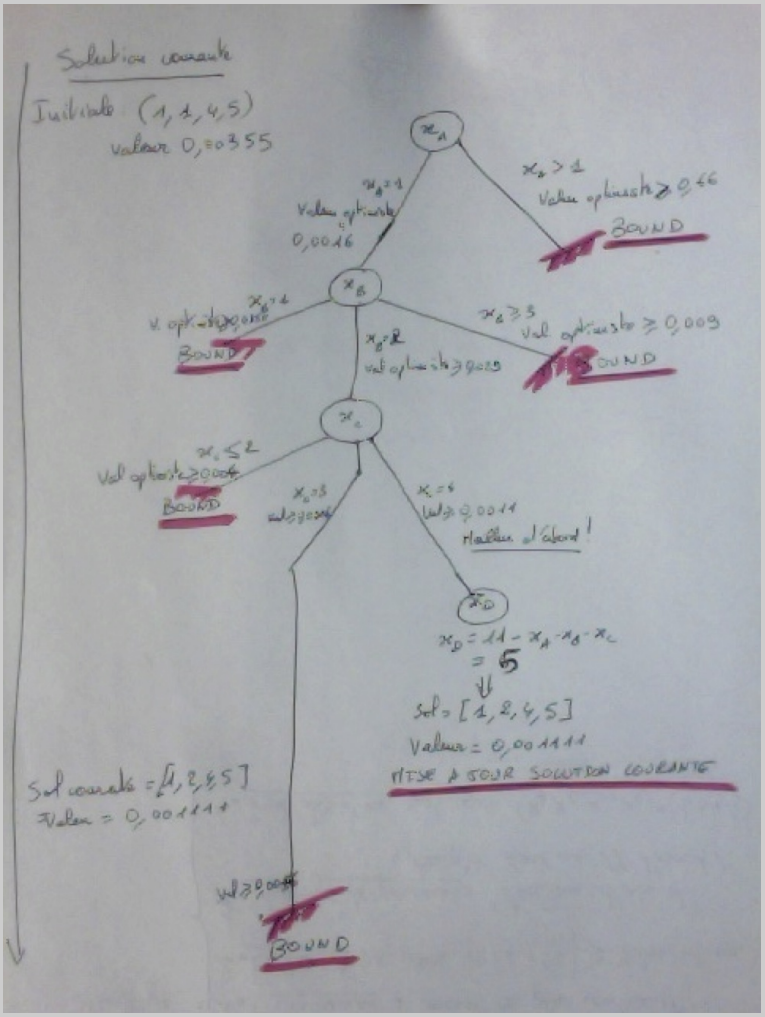
2. $(1, ?, ?, ?)$: Branch x_B : $x_B = 2$, valeur optimiste $\geq 0,0029$; pour $x_B \leq 1$, valeur optimiste = sol. cour. $\Longrightarrow$ BOUND
   pour $x_B \geq 3$, valeur optimiste $> 0,009 \Longrightarrow$ BOUND

3. (1,2, ?, ?) : Branch x_C : $x_C = 3$, valeur optimiste $\geq 0,0029$;
   : $x_C = 4$, valeur optimiste $\geq 0,0029$: supposons qu'on branche d'abord
   pour $x_C \leq 2$, valeur optimiste $> 0,004 \Longrightarrow$ BOUND
   pour $x_C \geq 5$, valeur optimiste $> 0,0038 \Longrightarrow$ BOUND

4. (1,2,3, ?) Branch x_D : On a $x_D = 11 - 1 - 2 - 3 = 5 \Longrightarrow$ solution (1,2,3,5) de valeur 0,0029 $\Longrightarrow$ améliore la solution courante, donc mise à jour !

5. (1,2,4, ?) valeur optimiste = sol. cour. $\Longrightarrow$ BOUND
   (en fait donne la solution (1,2,4,4) de même valeur 0,0029 que la solution courante).

Donc (1,2,3,5) est une solution optimale (l'autre solution optimale est (1,2,4,4)).



Question 2 Plus généralement, on veut distribuer d députés dans les n premiers districts en minimisant l'écart maximum de représentation par personne par rapport à l'idéal de un pour cent, avec la contrainte d'au moins un député par district. Proposer une carcatérisation récursive de la valeur minimale de cet écart maximum. Combien coûterait en mémoire et en ordre un programme qui implémenterait cette caractérisation ?

m_i le maximum de députés admissibles dans le district i ; comme il faut un député au moins par district, $m_i \leq d-(i-1)$. On a donc :

$$\text{Ecart}(d,n) = \min_{x=1,\ldots,d-n+1} \max\left(\left|0,01 - \frac{x}{p_n}\right|, \text{Ecart}(d-x, n-1)\right)$$

Le calcul du tableau Ecart par indice n croissant évite les calculs redondants. Il suffit de mémoriser dans un tableau $\text{X}(d,n)$ le nombre de députés dans le district n correspondant à $\text{Ecart}(d,n)$ pour obtenir la solution optimale.

Une implantation directe du programme récursif avec mémoisation utilise en espace mémoire $\boxed{d \times n}$.

Chaque appel demande $O(d)$ opérations, donc $\boxed{O\left(nd^2\right)}$ pour les au plus nd appels.

Plus précisément, en supposant les appels récursifs mémoïsés, le coût de l'appel $\text{Ecart}(k,i)$ en nombre de valeurs absolues, est $k-i+1$. De plus l'appel (k,i) est fait ssi $1 \leq i \leq n$ et, puisqu'il faut au moins par district, $i \leq k$ et, puisqu'il faut au moins 1 député par district pour les $n-i$ districts au dessus de i : $k \leq d-(n-i)$. D'où le nombre total de valeurs absolues :

$$\sum_{i=1}^{n} \sum_{k=i}^{d-n+i} k-i+1 = \sum_{i=1}^{n} \sum_{j=1}^{d-n+1} j = \frac{1}{2} n(d-n+1)(d-n+2),$$

soit un coût $\Theta\left((d-n)^2 n\right)$.

Remarque : Avec un clacul itératif par ligne (pour $i=1$ à n) et par nombre de député décroissant : pour $k = d-n+i$ à i, on peut faire le calcul en place en utilisant un seul tableau de taille n.

Remarque. Si pour le x minimum on a $\left|0,01 - \frac{x}{p_n}\right| > \text{Ecart}(d-x, n-1)$, alors il est suffisant mais non nécessaire d'avoir une sous-solution *optimale* pour les $d-x$ députés restants dans les $n-1$ circonscriptions restantes. Autrement dit, l'équation de Bellman impose de prendre une sous-solution optimale, donc ne considère pas nécessairement ici *toutes* les solutions optimales ; par contre, bien sûr, elle garantit qu'il n'existe pas de solution strictement meilleure. Ici, Bellman limite donc le champ de l'espace des solutions analysées : il suffit de considérer $n.d$ possibilités (le tableau des $\phi(i,j)$) ; on peut rater des solutions optimales qui ne vérifieraient pas l'hypothèse de sous-problème optimal ; mais on n'est sûr de ne pas rater de solutions meilleures.

B&B avec espace mémoire borné et cache (30')

Pour atteindre le plus rapidement possible une solution optimale et donc l'arborescence critique, l'ordre de parcours de la collection des sous-problèmes à explorer joue un rôle curcial. Cet ordre de parcours est défini par les méthodes `add()` et `pop()` de la collection, qui respectivement y ajoute et en extrait un élément.
Dans toute la suite on suppose que les collections `Stack` et `Queue` et `PriorityQueue` sont fournies et implémentent respectivement :

— pile (`Stack`) : l'élement extrait est le dernier ajouté (LIFO) ;
— file (`Queue`) : l'élement extrait est le premier ajouté (FIFO) ;
— file de priorité (`PriorityQueue`) : l'élément extrait est celui de plus forte priorité.

Question 3 (10') Stockage des nœuds à explorer dans une file de priorité ;

1. Entre deux sous-problèmes d'évaluations optimistes différentes et de même profondeur, lequel explorer en priorité ?
2. Entre deux sous-problèmes de même évaluation optimiste et de profondeurs différentes, lequel explorer en priorité ?
3. Quelle priorité utiliser lorsqu'on stocke les nœuds dans une file de priorité ?
4. Quelle place mémoire est requise en pire cas pour une file de priorité ?

1. Celui de meilleure évaluation optimiste
2. Celui de plus grande profondeur, a priori le plus proche d'une solution optimale
3. Réponse : compromis entre bonne évaluation optimiste et grande profondeur : par exemple priorité au nœud de meilleure évaluation (souvent les plus hauts dans l'arbre) le plus profond (souvent de moins bonne évaluation) ; ou une pondération entre ces deux critères.
4. En pire cas, parcours en largeur, donc mémoire = O(# éléments) : 2^n pour le sac à dos binaire par exemple.

Question 4 (15') **B&B avec contrainte mémoire et localité**. Soit T la taille mémoire requise pour stocker un nœud de l'arbre du B&B et soit d la profondeur maximum de l'arbre.
Pour s'exécuter dans un espace mémoire limité de taille M, (on suppose [1] $d \times T \ll M$s), on procède comme suit :

— Quand l'arbre contient moins de $M/T - d$ nœuds, l'ajout et le retrait de nœuds est géré par une file de priorité ;
— quand l'arbre contient plus de $M/T - d$ nœuds, l'ajout et le retrait de nœuds est géré par une pile (profondeur).

1. Implémenter une telle collection, à savoir l'initialisation (constructeur), la fermeture (destructeur), les méthodes `add`, `top`, `pop`, `estVide`.
2. Sur le modèle CO (cache de taille Z et lignes de cache de taille L remplacées par politique LRU), comment choisir M pour limiter le nombre de défauts de cache du B&B et quel est ce nombre ?
3. L'appel système `getpagefaults` donne le nombre de défauts de page depuis l'initialisation du sytème. Comment implanter l'algorithme ci-dessus en utilisant `getpagefaults` sans connaître Z, L ?

1. On implante une collection en implantant les méthodes `add` et `pop` en utilisant une file de priorité tant qu'il reste suffisamment de place puis une pile.

1. Cette hypothèse est raisonnable sinon même un parcours en profondeur de l'arbre est impossible.

```
class C extends Collection<Noeud> {
   PriorityQueue<Noeud> fileprio ; // pour le parcours par priorité
   Stack<Noeud> pile ; // Pour le parcours en profondeur
   int nbnoeud  ; // nombre de noeud dans la collection
   int nbnoeudmaxfile ; // nombre de noeud maximum dans la file de priorite

   -- Constructeur, initialisation
   C(size_t M, size_t T, int d) {
      nbnoeudmaxfile  = M / T - d ; -- on garde la place pour stocker au plus d noeuds
         dans la pile.
      pile  = new Stack<Noeud>(); //
      fileprio  = new PriorityQueue<Noeud>();
      nbnoeud  = 0 ;
   }

   void ~C() { // destructeur de la collection
      delete pile ;
      delete file ;
   }
   void add(Noeud& n) {
      if ( nbnoeud >= nbnoeudmaxfile ) pile.push( n ) ;
      else fileprio.add( n ) ;
      nbnoeud += 1 ;
   }

   Noeud& top() {
      if (nbnoeud <=0) { throw error() ; }
      if ( nbnoeud >= nbnoeudmaxfile ) return pile.top() ;
      else return fileprio.top() ;
   }

   void pop() {
      if (nbnoeud <=0) { throw error() ; }
      nbnoeud -= 1 ;
      if ( nbnoeud >= nbnoeudmaxfile ) return pile.pop( n ) ;
      else fileprio.pop( n ) ;
   }
}
```

2. On choisit $M = Z/2$. Le nombre de défauts est N / L.
3. Avec `getpagefaults` on mesure le nombre de défauts de page avant et après extraction d'un nœud de la file de priorité : si l'extraction génère un défaut de page, c'est que la file ne tient pas dans une page ; on passe alors à une pile. La méthode `pop` s'écrit simplement :

```
   void pop() { // Lors du constructeur nbnoeudmaxfile a été initialisé à MAXINT
      if (nbnoeud <=0) { throw error() ; }
      nbnoeud -= 1 ;
      if (nbnoeudmaxfile == MAXINT )
      { size_t a=getpagefaults() ;
        fileprio.pop( n ) ;
        if (a < getpagefaults()) nbnoeudmaxfile = nbboeud ;
        return ;
      }
      if ( nbnoeud >= nbnoeudmaxfile ) return pile.pop( n ) ;
      else fileprio.pop( n ) ;
   }
```

Remarque : Plus généralement, on peut précalculer la taille de cache Z par une recherche binaire avec $O(\log^2 Z)$ appels à `getpagefaults`. On initialise Z à 1 (ou plus si on a une idée de la taille de cache) et on alloue un tableau T de taille Z que l'on parcourt par indice croissant. On stocke `a = getpagefaults()`, on reparcourt le tableau dans l'autre sens : si `getpagefaults() == a`, on recommence en doublant Z ; sinon `getpagefaults() > a`, alors la taille de cache est entre $Z/2$ et Z. Avec une dichotomie sur le même principe on affine la valeur de la taille de cache Z.

Question 5 (15') **B&B anytime**.

1. On stoppe l'algorithme en cours d'exécution ; donner un algorithme qui calcule l'écart maximal de la valeur de la solution courante avec la valeur optimale.
2. Donner un algorithme qui donne une solution à un écart au plus e de l'optimum et qui explore au plus 2 fois plus de nœuds que nécessaire avec le même parcours.

1. On calcule la valeur optimiste la meilleure de tous les nœuds de la file et de la pile :

```
Val valOptimiste()
{ Noeud* ptcour = fileprio.begin() ; // Itérateur sur le file
  Val r = (ptcour++)->val;
  while (ptcour != fileprio.end() ) {// parcours de la file
  { Val cour = ptcour -> val ;
    if cour.EstMeilleur (r) r = cour ;
    ++ptcour ;
  }
  ptrcour = pile.begin();
  while (ptcour !- pile.end()) // parcours de la pile
  { Val cour = ptcour -> val ;
    if cour.EstMeilleur (r) r = cour ;
    ++ptcour ;
  }
  return r ;
}
```

2. On compte le nombre de `pop()` depuis le début. Si ce nombre est une puissance de 2, on lance l'algorithme `valOptimiste` et on s'arrête si l'écart avec `solcour.eval()`.

Sac à dos binaire

On considère un problème de sac à dos binaire : maximiser la valeur du sac de volume V en le remplissant avec des objets choisis parmi n ; pour $i = 0 \dots n-1$, l'objet i est de volume v_i et de valeur g_i.

Question 6 Instancier l'algorithme *branch&bound* générique pour résoudre le sac à dos binaire. Expliciter : la représentation d'un sous-problème ; la fonction d'évaluation optimiste.

- sous-problème : classes avec 4 attributs : `static int k; boolean x[0..n-1] ; int v; int g ;`
  - k est l'indice du prochain objet à mettre dans le sac (init 0) ;
  - Pour $0 \leq i < k$: $x[i]$ vaut 1 ssi on a pris l'objet i (init 0) ;
  - $r = \sum_{i=0}^{k-1} v_i \times x[i]$ est le volume restant (init V) ;
  - $g = \sum_{i=0}^{k-1} g_i \times x[i]$ est la valeur actuelle (init 0) ;
  - Constructeurs : initialise à un sac vide et constructeur de recopie.
- branch : un nœud (k, x, r, g) a deux fils : $(k+1, x', r, g)$ et $(k+1, x'', r - v[k], g + g[k])$ avec x' (resp. x'') a les mêmes composantes que x sauf $x[k]$ qui vaut 0 (resp. 1).
- bound : on suppose les objets triés par gain volumique décroissant (pré-tri). Plusieurs choix sont possibles :
  - bourrage avec l'objet x_k de gain volumique maximal parmi ceux restants à considérer : évaluation optimiste $(k, x, v, g) = g + \lfloor \frac{v}{v_k} \times g_k \rfloor$;
  - bourrage avec l'objet x_j de gain volumique maximal parmi ceux d'indice $\geq k$ et de volume inférieur à r : soit $j = \min_{i=k}^{n} \{i : v_i \leq r\}$, alors évaluation optimiste $(k, x, v, g) = g + \lfloor \frac{v}{v_j} \times g_j \rfloor$.
    Le coût de ce calcul est naïvement $O(n)$ en pire cas. On le réduit à un coût amorti $O(1)$ avec un pré-calcul en $O(n^3)$ du tableau $S[k, j] = \min_{i=k}^{n} \{i : v[i] \leq v[j]\}$ comme suit :
    1. pré-calcul du tableau S en $O(n^3)$
    2. pré-calcul un tableau T des indices de 0 à n triés par $v[i]$ décroissant en $O(n \log n)$: $v[T[1]] \leq v[T[2]] \leq \dots \leq v[T[n]]$.

3. on utilise la variable globale J, entier qui croît de 0 à n : à chaque étape, J est le plus petit indice tel que $v[T[J]] \leq r$.
   A chaque mise à jour de r, on met à jour J en l'incrémentant de 1 tant que $v[T[J]] > r$;
   L coût total de maintien de J pour une configuration du sac avec les n objets est donc $O(n)$, soit un coût amorti $O(1)$ à chaque pas.
4. $[k, J]$ vaut alors $\min_{i=k}^{n} \{i : v_i \leq v[J] \leq r\}$. L'évaluation optimiste est donc calculée en $O(1)$ par $g + \lfloor \frac{v}{v_{S[k,J]}} \times g_{S[k,J]} \rfloor$.

— solution relaxée : soit j l'objet de gain maximal qui rentre dans le sac (calculé en O(1) comme ci-dessus). On prend les i objets consécutifs à partir de j tant qu'ils rentrent, i.e. les objets $j, j+1, \ldots, j+i-1$; puis on met la fraction de $j+i$. On peut calculer i en moins de $\log_2 n$ comparaisons par recherche dichotomique. Il suffit de précalculer les préfixes sommes des volumes $\sigma_j = \sum_{i=0}^{n-1} v_j$. Puis, si r est le volume restant, on calcule l'indice $i + j$ en cherchant par dichotomie le volume virtuel $\sigma_j + r$ dans le sous-tableau trié $\sigma[j \ldots n-1]$.

1. pré-calcul du tableau $\sigma_j = \sum_{i=0}^{j} v_i$ en $O(n)$, donc O(1) amorti ;
2. pré-calcul du tableau $\gamma_j = \sum_{i=0}^{j} g_i$ en $O(n)$, donc O(1) amorti ;
3. calcul de l'indice j de l'objet restant de gain maximal rentrant dans le sac comme ci-dessus en O(1) ;
4. soit $i \geq j$ tel que $\sigma_{i-1} \leq \sigma_{j-1} + r < \sigma_i$, calculé par dichotomie en $O(\log n)$;
5. l'évaluation optimiste est alors $= g + \sum_{l=j}^{i-1} g_l + \lfloor \frac{r - \sum_{l=j}^{i-1} v_l}{v_i} \times g_i \rfloor = g + \gamma_{i-1} - \gamma_{j-1} + \lfloor \frac{r - \sigma_{i-1} + \sigma_{j-1}}{v_i} \times g_i \rfloor$ qui est calculée en O(1).

Le coût en pire cas est donc $\log_2 n$, en pratique on peut le considérer $O(1)$.

Question 7 La collection est une file (FIFO) : analyser temps d'exécution, place mémoire et localité en pire cas.

En pire cas, il n'y a aucune élagation et l'arbrescence est explorée entièrement : les feuilles sont les 2^n remplissages possibles du sac.
Nombre d'opérations en pire cas majoré par $= O(n2^n)$ (chaque configuration est construite en $O(n)$).

Une implémentation FIF0 plus fine (disons avec une "liste"), évite l'allocation et le stockage du tableau de taille n à chaque nœud. Par exemple, on chaîne en arbre les configurations des remplissages du sac entre elles (la racine est aucune variable fixée) : on ajoute un nœud interne en O(1) que lorsqu'on fixe une variable à 1. Le coût d'un nœud = allocation du nœud + mises à jour évaluation optimiste et volume restant à partir du père. On a alors : $T(n) = 2T(n-1) + \Theta(1)$; d'où $T(n) = \Theta(2^n)$.

De plus, en analysant plus finement, le coût est déséquilibré selon les nœuds.

— Le fils avec $x_n = 0$ n'a pas besoin d'être alloué ni d'être mis à jour : le pointeur vers la configuration (dans l'arbre des configurations), le volume restant et l'évaluation optimiste sont les mêmes que le père : il suffit donc d'incrémenter de 1 l'indice du prochain élément à considérer dans l'élément en tête de la liste FIFO.
— Le fils $x_n = 1$ a lui besoin d'être alloué et mis à jour (coût O(1) avec le chainage vers le père)

Avec l'élagage, il y a au plus 2^{n-1} nœuds dans chaque sous-arbre. Il y a donc au plus $T(n) = 2^{n-1}$ allocations, calculs d'évaluation optimiste et calculs de volume restant. Le coût est donc $O(2^n)$.
Espace mémoire requis $= \Theta(2^n)$: le parcours de l'arborescence, dont les feuilles sont les 2^n configurations, est fait en largeur.

Question 8 La collection est une pile (LIFO) : analyser temps d'exécution, place mémoire et localité en pire cas.

Le nombre d'opérations reste le même en pire cas $= \Theta(2^n)$ (aucun élagage).
Espace mémoire requis : exploration en profondeur de l'arborescence, donc au plus n nœuds en mémoire de coût $O(n)$ soit $O(n^2)$.
Remarque : en chaînant les configurations entre elles (le nœud fils a un pointeur sur le père), on peut réduire à un stockage de $O(1)$ par nœud soit un coût mémoire $O(n)$ au total (idem pour une implantation récursive avec backtrack).

Voyageur de commerce (TSP)

Donner le principe d'un algorithme *branch&bound* pour résoudre le problème du TSP : donner le principe des méthodes de branchement et d'évaluation optimiste (on ne demande pas de programme ici).

cf `2015-2016-AOD-seance2-slides.pdf`