P2 – Tas de Fibonacci

Une *file de priorité* est une structure de données utilisée pour stocker des objets avec une *valeur de priorité* (un nombre en virgule flottante) et qui supporte les opérations suivantes :

vide() Renvoie une file de priorité vide.

insère(file, objet, priorité) Ajoute un objet à la file, avec sa valeur de priorité.

dépile(file) Supprime l'objet dont la valeur de priorité est la plus haute dans la file, et le renvoie ainsi que sa priorité.

augmente(file, objet, priorité) Mettre à jour la valeur de priorité d'un objet, avec la condition que la nouvelle valeur de priorité doit être plus grande que l'ancienne.

Question 0. Donner le pseudo-code de l'algorithme de Dijkstra pour calculer la plus courte distance d'un nœud *source* à un nœud *destination* dans un graphe avec des poids positifs ou nuls. Le pseudo-code devra utiliser une file de priorité avec les opérations ci-dessus.

Quelle est la complexité en terme du nombre n de nœuds et m d'arêtes du graphe, ainsi que de la complexité des opérations de la file de priorité utilisée ?

Question 1. Quelle est la complexité de l'algorithme de Dijkstra si on utilise un tas binaire ?

Un *tas de Fibonacci* est une structure de données complexe utilisée pour implémenter des files de priorité. Formellement, c'est une collection de t arbres (non nécessairement binaires) dont les nœuds correspondent aux n objets à stocker. Chaque arbre vérifie la *condition de tas de priorité* : la priorité d'un nœud est toujours supérieure ou égale à la priorité de ses descendants. De plus, on mémorise un certain nombre d'informations supplémentaires :

- Chaque nœud d'un arbre a une *marque* qui est une variable booléenne, initialement mise à Faux, et mise à Vrai si le nœud *a perdu un enfant depuis la dernière fois qu'il a changé de parent.*
- Une variable spéciale *max* indique quel arbre a le nœud racine avec la plus haute valeur de priorité.
- Le nombre t d'arbres et le nombre $\delta(u)$ d'enfants de chaque nœud est également gardé en mémoire.

Question 2. Proposer une manière d'implémenter l'opération vide le plus simplement possible. Proposer une implémentation la plus simple possible de l'opération insère en $O(1)$.

Question 3. On associe à chaque tas de Fibonacci H un *potentiel* $\phi(H) \coloneqq \gamma \times (t + 2m)$ où t est le nombre d'arbres dans le tas, m le nombre de nœuds marqués à Vrai et γ une constante positive que l'on définira plus loin.

Pour une opération x qui transforme un tas de Fibonacci H en un nouveau tas de Fibonacci $x(H)$ avec coût de calcul $c_x(H)$, on considère $\hat{c}_x(H) \coloneqq c_x(H) + \phi(x(H)) - \phi(H)$. Montrer que pour toute séquence d'opérations $x_1 \dots x_k$ à partir du tas de Fibonacci vide H_0, avec $H_i \coloneqq x_i(H_{i-1})$: $\frac{1}{k}\sum_{i=1}^{k} c_{x_i}(H_{i-1}) \leq \frac{1}{k}\sum_{i=1}^{k} \hat{c}_{x_i}(H_{i-1})$. On appellera $\hat{c}_x(H)$ le *coût amorti* de l'opération x.

Question 4. Dans les tas de Fibonacci, dépile fonctionne comme suit : on enlève le nœud pointé par le pointeur *max* et on fait de chacun de ses enfants une nouvelle racine d'un arbre. Ensuite, on s'assure que chaque racine r a un nombre d'enfants $\delta(r)$ distinct : pour ce faire, chaque fois que deux racines ont le même nombre d'enfants, l'une devient le fils de l'autre (en respectant la condition de tas de priorité), et on répète jusqu'à ce que toutes les racines aient des degrés distincts. Pendant ces opérations, on met à jour la marque quand c'est nécessaire.

Montrer que, en choisissant une valeur appropriée de γ, la *coût amorti* de dépile dans un tas de Fibonacci H est en : $O\left(\max\left(\max_{u \text{ nœud de } H} \delta(u), \max_{u \text{ nœud de dépile}(H)} \delta(u)\right)\right)$.

Suite des questions

Question 5. Dans les tas de Fibonacci, augmente fonctionne comme suit : si, après mise à jour des priorités sur un nœud u, la condition de tas de priorité est violée, u est détaché de son parent p et devient la racine d'un nouvel arbre. On procède ensuite en cascade sur le nœud p : si le nœud p avait déjà sa marque à Vrai, on le détache de son propre parent p' et il devient la racine d'un nouvel arbre ; on continue récursivement sur p'.

Montrer qu'en choisissant une valeur appropriée pour γ (compatible avec le choix déjà fait), le coût amorti de augmente est un $O(1)$. On pourra introduire le nombre de fois que la procédure de cascade a été appelée.

Question 6. On veut maintenant montrer que le degré maximal d'un nœud dans un tas de Fibonacci est en $O(\log n)$.

a) Montrer que pour tout nœud u avec enfants $u_1, u_2, \ldots, u_k$ ordonnés dans l'ordre chronologique de leur ajout comme enfant de u, pour tout $1 \leq i \leq k$, $\delta(u_i) \geq i - 2$.

b) Soit F_k le k-ième terme de la suite de Fibonacci : $\begin{cases} F_0 = 0 \quad F_1 = 1 \\ F_k = F_{k-1} + F_{k-2} \quad \text{pour } k \geq 2. \end{cases}$. Montrer que pour tout nœud u d'un tas de Fibonacci, la taille du sous-arbre enraciné en u est $\geq F_{\delta(u)+2}$.

c) Montrer que, pour tout k, $F_{k+2} \geq \left(\frac{1+\sqrt{5}}{2}\right)^k$ et conclure.

Question 7. Quelle est la complexité *amortie* de l'algorithme de Dijsktra (c.-à-d., en remplaçant les coûts réels des opérations par leurs coûts amortis) si on implémente la file de priorité avec un tas de Fibonacci ? Comparer avec le tas binaire.

Corrigé

Question 0. On suppose pour simplifier que les nœuds du graphe sont numérotés par des entiers consécutifs à partir de 0.

```
Fonction Dijkstra(G, source, destination)
    n ← nombre de nœuds de G;
    f ← vide();
    d ← tableau de n éléments;
    pour u nœud de G faire
        insère(f, u, −∞);
        d[u] ← ∞;
    augmente(source, 0);
    d[source] ← 0;
    tant que (u, p) ← dépile(f) faire
        pour (u, v, w) ∈ G faire
            si −p + w ≤ d[v] alors
                d[v] ← −p + w;
                augmente(f, v, −d[v]);
    retourner d[destination];
```

Si les complexités de insère, augmente et dépile sont respectivement de ι, α et δ, la complexité est de $O(n \times (\iota + \delta) + m \times \alpha)$.

On peut bien sûr accélérer un peu cet algorithme en arrêtant la boucle dès que la distance à *destination* a été déterminée, sans changer la complexité asymptotique.

Question 1. On a $\iota = \delta = \alpha = O(\log n)$ dans un tas binaire. En effet, insère et dépile sont des opérations classiques ; pour augmente on peut par exemple supprimer puis rajouter l'élement avec la même priorité – il est nécessaire d'avoir une structure de données auxiliaire (comme un simple tableau) pour savoir où chaque nœud se trouve dans le tas afin de trouver l'élément à supprimer en $O(1)$.

On obtient donc une complexité en $O((n+m)\log n)$ pour la complexité de l'algorithme de Dijkstra.

Question 2. Pour vide(), il suffit de retourner un tas avec $t = 0$ arbre et *max* non positionné.

Pour insère(f, o, p), afin de garantir une insertion simple en $O(1)$, on peut juste ajouter o à f comme une nouvelle racine d'un arbre formé de ce seul élément, avec la *marque* mise à Faux, et *max* pointant vers ce nouvel arbre si et seulement si la priorité p est supérieure à la priorité de l'arbre actuellement pointé par *max*.

Question 3. On a :

$$\begin{aligned}\sum_{i=1}^{k} \hat{c}_{x_i}(H_{i-1}) &= \sum_{i=1}^{k} c_{x_i}(H_{i-1}) + \sum_{i=1}^{k} \phi(H_i) - \sum_{i=0}^{k-1} \phi(H_i) \\ &= \sum_{i=1}^{k} c_{x_i}(H_{i-1}) + \phi(H_k) - \phi(H_0) \\ &\geq \sum_{i=1}^{k} c_{x_i}(H_{i-1})\end{aligned}$$

puisque $\phi(H_0) = \gamma \times (0 + 2 \times 0) = 0$ et ϕ est une fonction à valeur positive ou nulle. On obtient le résultat demandé en divisant les deux membres de l'inégalité par k.

Question 4. On pose $D = \max\left(\max_{u \text{ nœud de } H} \delta(u), \max_{u \text{ nœud de dépile}(H)} \delta(u)\right)$.

Appelons u le nœud pointé par *max*. Enlever ce nœud et faire de ses enfants de nouvelles racines se fait en $O(\delta(u))$. On énumère ensuite les degrés des racines en $O(t + \max_{u \text{ nœud de } H} \delta(u)) = O(t + D)$ en stockant par exemple dans un tableau indexé par le degré les nœuds ayant un degré donné, ce tableau ne nécessitant de considérer que $O(t + D)$ degrés différents. Chaque fois que deux racines ont le même degré, on rend l'une fils de l'autre en $O(1)$ en comparant leurs priorités. On peut devoir répéter cette opération, mais comme à chaque fois le nombre de racines décroît d'une unité, on réalise cette opération au plus $O(t + \delta(u)) = O(t + D)$ fois. On a donc un coût (non amorti) de chaque appel à dépile en $O(t + D)$. Soit c_1 une constante telle que, pour un tas de Fibonacci suffisamment grand, le coût (non amorti) est $\leq c_1(t + D)$.

Le coût amorti est $\leq c_1(t + D) + \gamma(D + 1 + 2m - t - 2m)$: on a au final au plus $D + 1$ arbres et on n'ajoute pas de nouvelle marque à Vrai (on peut éventuellement en enlever, mais l'inégalité reste correcte). On choisit $\gamma \geq c_1$ de sorte que $t(c_1 - \gamma) \leq 0$. On a alors un coût amorti $\leq c_1 D + \gamma(D + 1)$, soit en $O(D)$.

Question 5. Détacher un nœud de son parent et le faire devenir la racine d'un nouvel arbre peut se faire en $O(1)$. Cette opération peut être faite un certain nombre ℓ de fois, mais à chaque fois (sauf éventuellement la première, où le nœud n'est pas nécessairement marqué), une marque passe à Faux ; une fois qu'on atteint un parent que l'on ne détache pas, la marque de ce parent passe en revanche à Vrai. On a donc un coût amorti, pour un tas de Fibonacci suffisamment grand qui est $\leq c_2\ell + \gamma(t + \ell + 2(m - (\ell + 1) + 1) - t - 2m) = c_2\ell + \gamma(4 - \ell)$ pour une certaine constante c_2. On prend $\gamma \geq c_2$, ce qui est compatible avec le choix $\gamma \geq c_1$ fait précédemment. On obtient un coût amorti $\leq 4\gamma$, et donc en $O(1)$.

Question 6.

a) On remarque que la seule opération qui ajoute un nœud u_i comme enfant d'un autre nœud u est dépile et uniquement si l'ancien degré du nœud u avant cet ajout (qui est précisément $i-1$) était identique au degré du nœud u_i au moment où il a été ajouté. Depuis cet ajout, u_i a perdu au plus un enfant lors de l'exécution de augmente – si u_i avait perdu plus d'un enfant, augmente l'aurait détaché de u (et u_i n'a pas pu gagner d'enfant, aucune opération n'ajoute à un enfant à un nœud non-racine). On a donc $\delta(u_i) \in \{i-2, i-1\}$ et, en particulier, $\delta(u_i) \geq i-2$.

b) On pose s_k la taille minimale du sous-arbre enraciné en un nœud ayant k enfants. Clairement la fonction $k \mapsto s_k$ est croissante.
Soit u un nœud arbitraire avec comme enfants $u_1, \ldots, u_k$. On a :

$$\begin{aligned} s_{\delta(u)} &\geq 1 + \sum_{i=1}^{k} s_{\delta(u_i)} \\ &\geq 2 + \sum_{i=2}^{k} s_{i-2} \end{aligned}$$

On montre par récurrence sur k que $s_k \geq F_{k+2}$. Les cas de base sont clairs : $s_0 = 1 = F_2$ et $s_1 = 2 = F_3$. Pour le cas inductif, on suppose que $k \geq 2$ et que l'inégalité est vraie pour tout $i < k$. On a :

$$s_k \geq 2 + \sum_{i=2}^{k} s_{i-2} \geq 2 + \sum_{i=2}^{k} F_i = 1 + \sum_{i=0}^{k} F_i = F_{k+2}$$

(Cette dernière égalité peut facilement se montrer par récurrence sur k).

c) On démontre l'inégalité par récurrence sur k.
- Pour $k = 0$, $F_{k+2} = F_2 = 1 \geq 1 = \left(\frac{1+\sqrt{5}}{2}\right)^k$.
- Pour $k = 1$, $F_{k+2} = F_3 = 2 \geq \left(\frac{1+\sqrt{5}}{2}\right)^k$ vu que $\sqrt{5} < 3$, c.-à-d., $\frac{1+\sqrt{5}}{2} < 2$.
- Soit $k \geq 2$ et supposons l'inégalité vraie aux rangs $k-2$ et $k-1$. On a :

$$F_{k+2} = F_{k+1} + F_k \geq \left(\frac{1+\sqrt{5}}{2}\right)^{k-2} \times \left(\frac{1+\sqrt{5}}{2} + 1\right)$$

Or $\left(\frac{1+\sqrt{5}}{2}\right)^2 = \frac{6+2\sqrt{5}}{4} = \frac{1+\sqrt{5}}{2} + 1$, d'où $F_{k+2} \geq \left(\frac{1+\sqrt{5}}{2}\right)^k$.

En conclusion, on a montré que la taille du sous-arbre enraciné en tout nœud u est $\geq \psi^{\delta(u)}$ où $\psi = \frac{1+\sqrt{5}}{2}$. En particulier, $\psi^{\delta(u)} \leq n$ et $\delta(u) \leq \log_\psi(n)$. Cela est vrai pour n'importe quel nœud u, en particulier pour le degré maximal, qui est donc en $O(\log n)$.

Question 7. En utilisant les résultats des questions précédentes, on a pour le coût amorti de insère, augmente et dépile respectivement $O(1)$, $O(1)$ et $O(\log n)$. On obtient donc une complexité amortie de $O(m + n \log n)$ ce qui est asymptotiquement mieux que le $O((n+m)\log n)$ obtenu par une implémentation de Dijkstra avec un tas binaire comme file de priorité ; la différence ne devient cependant visible que pour les graphes ayant beaucoup d'arêtes.

J5 – Arbres à boucs émissaires

Question 0.

(a) Rappeler ce qu'est un arbre binaire de recherche et comment un tel arbre peut être utilisé pour implémenter la structure de données de dictionnaire. On supposera que chaque clé est unique.

(b) Donner le pseudo-code de la fonction de recherche dans un arbre binaire de recherche. Quelle est sa complexité en pire cas ?

La *taille* d'un arbre binaire de recherche t est le nombre de nœuds qu'il contient. On la note $|t|$. Sa *hauteur* est le nombre maximal de nœuds sur un chemin allant directement de sa racine à l'une de ses feuilles. On la note $h(t)$.

Soit $\alpha \in \left[\frac{1}{2}, 1\right]$. On dit qu'un arbre est *α-équilibré en taille* si, pour chacun de ses sous-arbres t de fils gauche et droit g et d, on a :

$$|g| \leq \alpha\,|t| \qquad\qquad |d| \leq \alpha\,|t| \tag{1}$$

Question 1.

(a) Quels sont les arbres 1-équilibrés en taille ?

(b) Écrire le pseudo-code d'un algorithme permettant de construire un arbre de recherche $\frac{1}{2}$-équilibré en taille dont le contenu est donné par un tableau trié donné en entrée. Quelle est sa complexité en temps et en espace ?

On suppose maintenant que $\alpha \in \left]\frac{1}{2}, 1\right[$. On dit qu'un arbre t est *α-équilibré en hauteur* lorsque $h(t) \leq 1 + \frac{\ln|t|}{-\ln\alpha}$.

Question 2. Démontrer qu'un arbre non vide α-équilibré en taille est α-équilibré en hauteur.

Question 3. Donner la complexité de la recherche d'un élément dans un arbre α-équilibré en hauteur.

Pour insérer une nouvelle paire clef-valeur dans un arbre α-équilibré en hauteur, on commence par utiliser l'algorithme habituel. Si l'arbre n'est plus α-équilibré en hauteur, on cherche un *bouc émissaire* : il s'agit du plus proche ancêtre du nouveau nœud qui ne vérifie pas les inégalités (1). Puis, on remplace le sous-arbre correspondant au bouc émissaire par un arbre du même contenu construit avec l'algorithme de la question 1b.

Question 4.

(a) Un tel bouc émissaire existe-t-il toujours ?

(b) L'arbre ainsi obtenu est-il toujours α-équilibré en hauteur ?

(c) Quelle est sa complexité en temps et en espace, dans le pire et le meilleur cas ?

Pour faire une analyse plus fine de la complexité, on associe à chaque sous-arbre $t = N(g, k, v, d)$ la quantité $\Delta(t) = \max\{0, \mathrm{abs}(|g| - |d|) - 1\}$. De plus, on note $\Phi(t)$ la somme de tous les $\Delta(t')$ pour t' un sous-arbre (non nécessairement direct) de t.

Question 5.

(a) Donner une borne supérieure sur la variation de $\Phi(t)$ lors d'une insertion. Celle-ci dépendra, le cas échéant, de la taille du bouc émissaire.

(b) Quelle est la complexité globale en temps de l'insertion de N paires clef-valeur en partant d'un arbre vide ?

Corrigé

Question 0.

(a) Lorsqu'il est utilisé pour implémenter un dictionnaire, un arbre binaire de recherche est un arbre binaire dont les nœuds sont étiquetés par une clef et la valeur correspondante. Pour tout nœud interne $N(g, k, v, d)$ d'un arbre binaire de recherche, les nœuds du sous-arbre gauche g sont étiquetés par des clefs plus petites que k, et les nœuds du sous-arbre droit d sont étiquetés par des clefs plus grandes que k. Lorsque l'on veut lire ou modifier la valeur associée à une clef, on recherche le nœud correspondant en comparant les clefs avec la clef recherchée, et on y fait l'opération voulue. Lorsque l'on veut insérer une nouvelle valeur, on crée un nouveau nœud en bas de l'arbre, à la seule position possible.

Note : dans le pire cas (si les entrées sont insérées dans l'ordre), l'arbre devient déséquilibré, et les accès au dictionnaire sont en temps linéaire. Pour éviter ce problème et obtenir des temps d'accès logarithmiques, il faut équilibrer l'arbre. L'objectif de ce sujet est d'expliquer une méthode d'équilibrage.

(b)
```
Fonction cherche(t, k)
  Si t = N(g, k', v, d)
    Si k == k'
      Renvoyer v
    Si k < k'
      Renvoyer cherche(g, k')
    Si k > k'
      Renvoyer cherche(d, k')
  Sinon
    Renvoyer INTROUVABLE
```

Sa complexité est proportionnelle à la hauteur de l'arbre. Si l'arbre est déséquilibré, cela peut être linéaire en la taille de l'arbre.

Question 1.

(a) Bien sûr, on a toujours $|d| \leq |t|$ et $|g| \leq |t|$. Donc tout arbre binaire de recherche est 1-équilibré.

(b) Pour qu'un arbre soit 1/2-équilibré, il faut que :

$$|g| \leq \frac{1}{2}|t| = \frac{1}{2}(1 + |g| + |d|)$$

C'est-à-dire :

$$|g| \leq 1 + |d|$$

Et symétriquement :

$$|d| \leq 1 + |g|$$

On démontre facilement que ces deux conditions constituent une condition nécessaire et suffisante. Pour construire l'arbre à partir du tableau, il faut donc mettre une clef médiane à la racine de l'arbre, et recommencer récursivement pour les sous-arbres gauche et droit. On obtient donc le pseudo-code suivant :

```
Fonction construitArbre(tab, deb, fin)
  Si fin == deb
    Renvoyer Feuille
  Sinon
    Soit mid = (deb+fin)/2
    Renvoyer N(construitArbre(tab, deb, mid), tab[mid],
               construitArbre(tab, mid+1, fin))
```

Chaque appel récursif correspond exactement à une feuille ou un nœud. Puisqu'il en existe un nombre linéaire, la complexité en temps est linéaire. La complexité en mémoire, si on ne compte pas l'entrée et la sortie, est logarithmique (il s'agit de la taille de la pile, qui correspond à la hauteur de l'arbre).

Note : Il est important de ne pas copier le tableau, mais de réutiliser le même en passant des indices de début et fin. Sinon, la complexité devient $O(N \log N)$.

Question 2. On prouve cela par induction sur l'arbre. Si l'arbre est réduit à un nœud, alors $h(t) = 1$ et $|t| = 1$, et l'inégalité est vérifiée. Sinon, l'arbre est de la forme $N(g, k, v, d)$, avec :

$$|g| \leq \alpha\,|t| \qquad |d| \leq \alpha\,|t|$$
$$h(g) \leq 1 + \frac{\ln|g|}{-\ln\alpha} \qquad h(d) \leq 1 + \frac{\ln|d|}{-\ln\alpha}$$

Or :

$$\begin{aligned} h(t) &= 1 + \max\{h(g), h(d)\} \\ &\leq 2 + \frac{\ln\max\{|g|, |d|\}}{-\ln\alpha} \\ &\leq 2 + \frac{\ln(\alpha\,|t|)}{-\ln\alpha} \\ &= 1 + \frac{\ln|t|}{-\ln\alpha} \end{aligned}$$

Question 3. La profondeur d'un tel arbre étant logarithmique, la recherche d'un élément peut s'effectuer, par la fonction de la question 0, en temps logarithmique par rapport à sa taille.

Question 4. *Ce n'est pas demandé dans le sujet, mais voici le pseudo-code de l'algorithme en question. Note : il faut stocker la taille de l'arbre dans une variable globale* `t.taille` *pour ne pas avoir à la recalculer à chaque insertion.*

```
Fonction contenu(t, tab, deb)
  Si t = N(g, k, v, d)
    Soit fin = contenu(g, deb)
    t[fin] = (k, v)
    Renvoyer contenu(d, fin+1)
  Sinon // Feuille
    Renvoyer deb

Fonction taille(t)
  Si t = N(g, _, _, d)
    Renvoyer taille(g) + taille(d) + 1
  Sinon
    Renvoyer 0

Fonction insèreAux(t, k, v, h)
  Si t = F
     Si h < 0
       Renvoyer Déséquilibré(N(F, k, v, F), 1)
     Sinon
       Renvoyer Équilibré(N(F, k, v, F))
```

```
  Si t = N(g, k', v', d)
    Si k < k'
      resi = insèreAux(g, k, v, h-1)
      Si resi = Équilibré(g')
        Renvoyer Équilibré(N(g', k', v', d))
      Si resi = Déséquilibré(g', ng')
        d' = d
        nd' = taille(d')
    Sinon
      resi = insèreAux(d, k, v, h-1)
      Si resi = Équilibré(d')
        Renvoyer Équilibré(N(g, k', v', d'))
      Si resi = Déséquilibré(d', nd')
        g' = g
        ng' = taille(g')
    n = ng' + nd' + 1
    Si ng' > alpha * n Ou nd' > alpha * n
      tab = tableau de taille n
      contenu(N(g', k, v, d'), tab, 0)
      Renvoyer(Équilibré(construitArbre(tab, 0, n)))
    Sinon
      Renvoyer(Déséquilibré(N(g', k, v, d'), n))

Fonction insère(t, k, v)
  t.taille += 1
  t.arbre := insèreAux(t.arbre, k, v, 1-ln(t.taille)/ln(alpha))
```

(a) *Indication : remarquer que si l'insertion a cassé l'α-équilibre en hauteur, alors le nouveau nœud est forcément à la profondeur maximale, puisque la hauteur de l'arbre vient nécessairement de changer.*

Un tel bouc émissaire existe toujours. En effet, dans le cas contraire, en itérant la définition de l'α-équilibre en taille, on obtiendrait :

$$1 \leq \alpha^{h(t)-1} |t| < \alpha^{1+\frac{\ln|t|}{-\ln\alpha}-1} |t| = 1$$

Ce qui est contradictoire.

(b) S'il n'y a pas besoin de bouc émissaire, il n'y a rien à prouver. Sinon, soit t le bouc émissaire, avant qu'il ne soit rééquilibré. Puisque l'arbre était globalement α-équilibré en hauteur, le bouc émissaire contient un unique nœud de hauteur maximale : celui que nous venons d'insérer, et qui a cassé l'α-équilibre.

Par ailleurs, le bouc émissaire n'est pas α-équilibré en taille. On peut en déduire (comme à la question 1) que la différence de taille de ses deux enfants est plus grande ou égale à 2. Et avant l'insertion, cette différence était au moins égale à 1. Ceci prouve que, si l'on avait ignoré la valeur des clefs, le nouveau nœud aurait pu être placé à une position ne modifiant pas la hauteur de l'arbre bouc émissaire. Par conséquent, puisque le rééquilibrage remplace l'arbre bouc émissaire par un arbre de hauteur minimale pour sa taille, la hauteur du nouvel arbre est au plus égale à la hauteur du bouc émissaire *avant l'insertion*.

Puisque l'insertion n'augmente pas la hauteur de l'arbre, et que celui-ci était α-équilibré en hauteur avant l'insertion, il est bien α-équilibré en hauteur après l'insertion.

(c) S'il n'y a pas de bouc émissaire, alors la complexité est proportionnelle à la hauteur de l'arbre, c'est-à-dire logarithmique en sa taille. Si, au contraire, il y a eu rééquilibrage, alors il faut rajouter au terme logarithmique la taille du bouc émissaire, ce qui correspond au coût de son rééquilibrage.

Question 5.

(a) S'il n'y a pas de rééquilibrage, alors chaque $\Delta(t)$ ne peut varier que d'une unité au plus, et cette variation n'a lieu que pour les nœuds se trouvant sur le chemin de la racine au nouveau nœud. Étant donné que ce chemin est de longueur au plus $1 + \frac{\log|t|}{-\log\alpha}$, le potentiel $\Phi(t)$ n'augmentera pas plus que cette quantité.

Lors d'un rééquilibrage, on rajoute à cette quantité la variation de $\Phi(t)$ due à ce rééquilibrage. Après le rééquilibrage, tous les $\Delta(t')$ concernant des sous-arbres du sous-arbre reconstruit sont nuls.

Par ailleurs, si $t' = N(g, k, v, d)$ est un bouc émissaire, supposons, par exemple, $|g| > \alpha\,|t'|$. On a alors $|g| - |d| = 2\,|g| - |t'| + 1 > (2\alpha - 1)\,|t'| + 1$. Donc $\Delta(t') > (2\alpha - 1)\,|t'|$ (ceci étant vrai aussi lorsque $|d| > \alpha\,|t'|$).

Donc la variation de $\Phi(t)$ due au rééquilibrage est au plus $-(2\alpha - 1)\,|t'|$ si t' est le bouc émissaire.

Ainsi, lors d'une insertion faisant intervenir un rééquilibrage, la variation de $\Phi(t)$ est majorée par $1 + \frac{\log|t|}{-\log\alpha} - (2\alpha - 1)\,|t'|$, où t' est le bouc émissaire utilisé.

(b) On peut se servir de Φ comme une "réserve de temps" que l'on peut utiliser pour les insertions coûteuses. Ainsi, le coût d'une insertion est la somme du coût d'une insertion décrit dans la question 4c et de la variation de $\Phi(t)$, éventuellement multipliée par une constante. Si l'on choisit cette constante convenablement, le terme $(2\alpha - 1)\,|t'|$ de la variation de $\Phi(t)$ va compenser la reconstruction du bouc émissaire. Le coût résiduel de l'insertion d'un nouveau nœud est donc logarithmique, et la complexité totale associée à la création d'un arbre de taille N est donc $O(N \log N)$.

J3 – B-arbres

Question 0.

(a) Rappeler ce qu'est un arbre binaire de recherche.

(b) Généraliser cette notion pour permettre aux nœuds de l'arbre d'avoir plus de deux fils tout en permettant des recherches d'éléments efficaces.

(c) Dans le cas d'une grande quantité d'information, il est parfois nécessaire d'utiliser des supports de stockage dont le temps de réponse pour une lecture est élevé (disque dur, ...). Expliquer alors l'intérêt de la structure de donnée décrite dans la question (b).

Soit un entier $m \geq 3$. On appelle B-arbre un arbre tel que décrit à la question 0.c, et vérifiant de plus les invariants suivants :

— toutes les feuilles ont la même profondeur ;

— le nombre d'étiquettes de tout nœud est au plus $m - 1$;

— le nombre d'étiquettes de tout nœud non racine est au moins $\lceil \frac{m}{2} \rceil - 1$;

— la racine a au moins une étiquette.

Question 1. Donner le pseudo-code d'un algorithme de recherche *efficace* dans un B-arbre et en donner la complexité en temps en fonction de m et du nombre d'étiquettes stockées dans l'arbre.

Question 2.

(a) Proposer le pseudo-code d'un algorithme d'insertion d'une nouvelle entrée dans un B-arbre. Quelle est sa complexité en temps dans le pire cas ?

(b) Si on utilise cet algorithme pour insérer des étiquettes dans l'ordre croissant, combien d'étiquettes un nœud de l'arbre résultant contiendra-t-il, typiquement ?

(c) Proposer une variante de l'algorithme d'insertion qui permet un meilleur remplissage des nœuds dans le scénario de la question précédente. Quelle est sa complexité en temps dans le pire cas ?

Question 3. Proposer un algorithme de suppression d'une entrée dans un B-arbre. Quelle est sa complexité en temps ?

Question 4. Proposer un algorithme qui prend en paramètre deux B-arbres T_1 et T_2 tels que les clefs de T_1 sont toutes inférieures à celles de T_2, et qui renvoie un B-arbre contenant les étiquettes de T_1 et de T_2. Quelle est sa complexité en temps ?

Suite des questions

Question 5. Proposer une variante des algorithmes d'insertion et de suppression dans un B-arbre tels que le nombre de fils d'un nœud non racine est compris entre $\lceil \frac{2m-1}{3} \rceil$ (inclus) et m (inclus). Quelle contrainte sur le nombre de fils de la racine doit-on maintenant avoir ?

Question 6. Donner le pseudo-code des fonctions de la question 2.c, 3 et 4.

Corrigé

Question 0. (a) Un arbre binaire de recherche est un arbre binaire étiqueté par des éléments d'un ensemble totalement ordonné. Il vérifie la propriété suivante : si l'arbre contient un nœud $\texttt{Nœud}(g, x, d)$ (de sous-arbres gauche et droit g et d, et d'étiquette x), alors toutes les étiquettes des nœuds de g sont plus petites que x, et toutes les étiquettes des nœuds de d sont plus grandes que x. Dans le contexte de cette épreuve, on ignore le cas où plusieurs étiquettes stockées dans un même arbre pourraient être identiques (c'est impossible pour les dictionnaires).

(b) Si un nœud d'un tel arbre a plus de deux fils, alors il est naturel de demander que les étiquettes d'un fils soient toutes plus petites que celles de ses frères droits, et toutes plus grandes que celles de ses frères gauches.

Cependant, cela ne permet pas d'avoir un algorithme de recherche efficace : en effet, pour chercher une étiquette dans un arbre, il faut pouvoir déterminer facilement dans quel sous-arbre celle-ci devrait se trouver. Afin de résoudre ce problème et de permettre une recherche efficace, si un nœud a n fils, alors on stocke dans chaque nœud $n-1$ étiquettes qui permettent de « séparer » les sous-arbres. Si l'arbre est utilisé pour représenter un dictionnaire, alors on stocke aussi les valeurs associées à ces étiquettes dans les nœuds. Ainsi, si les sous-arbres sont $T_0, \ldots, T_{n-1}$ et les étiquettes stockées dans le nœud $x_0, \ldots, x_{n-2}$, alors que l'étiquette x_i soit plus grande que toutes les étiquettes du sous-arbre T_i et plus petite que toutes les étiquettes du sous-arbre T_{i+1}. On voit que, lorsque $n = 2$, cela correspond exactement au cas d'un arbre binaire de recherche. Il est alors facile, en regardant simplement le contenu d'un nœud, de savoir dans quel sous-arbre il faut continuer la recherche.

(c) Dans un arbre binaire de recherche, on doit faire un accès à la mémoire (donc au disque dur, si l'arbre est stocké dans un disque dur) pour chaque nœud menant au nœud recherché. Si l'arbre est bien équilibré, cela représente à peu près $\log_2 N$ accès mémoire.

Par contre, si le degré de branchement de l'arbre est plus élevé, on peut faire beaucoup moins d'accès. Par exemple, si chaque nœud interne a d fils, alors il faut faire environ $\log_d N$ accès mémoire. Si, par exemple, $d = 128$, cela représente environ 7 fois moins d'accès mémoire que pour un branchement binaire.

Bien sûr, cela vient avec une contrepartie : les nœuds sont plus gros, et il faut donc lire plus d'information à la fois pour chaque accès mémoire. Et, pour la même raison, la modification des nœuds est plus coûteuse. Il y a donc un compromis à chercher.

Question 1. On suppose qu'un nœud est représenté par quatre informations :

- le nombre de ses sous-arbres,
- un tableau de ses sous-arbres,
- un tableau des étiquettes qu'il contient,
- un tableau des valeurs associées à ces étiquettes.

On effectue la recherche dans un B-arbre de la façon suivante :

```
Fonction Cherche(x, nœud)
  i = Cherche_premier_supérieur(x, nœud.nombre_fils-1, nœud.étiquettes)
```

```
  Si i < nœud.nombre_fils-1 ET nœud.étiquettes[i] == x
    Renvoyer nœud.valeurs[i]
  Renvoyer Cherche(x, nœud.fils[i])
```

Où la fonction `Cherche_premier_supérieur(x, n, t)` prend en paramètre une étiquette `x`, un tableau trié `t` d'étiquettes et sa taille `n` et cherche dans `t` la première cellule dont le contenu est supérieur ou égal à `x`. Si une telle cellule est trouvée, alors cette fonction renvoie son indice ; sinon, elle renvoie `n`.

Naïvement, on pourrait implémenter la fonction `Cherche_premier_supérieur` avec une simple boucle :

```
Fonction Cherche_premier_supérieur(x, n, t)
  Pour i dans [0, n[
    Si t[i] >= x
      Renvoyer i
  Renvoyer n
```

Pour cette question, cette implémentation n'est pas suffisamment efficace. Mais on peut aussi utiliser une recherche dichotomique pour plus d'efficacité lorsque m est grand :

```
Fonction Cherche_premier_supérieur(x, n, t)
  a = 0, b = n
  Tant que b-a >= 1
    Si t[(a+b)/2] > x
      b = (a+b)/2
    Sinon
      a = (a+b)/2+1
  Renvoyer a
```

On obtient alors une complexité en temps de $O(p \log m)$, où p est la hauteur du nœud recherché. On sait que p est bornée par la profondeur (commune) des feuilles. On peut prouver facilement, par ailleurs, qu'un arbre dont les nœuds internes ont au moins $\lceil \frac{m}{2} \rceil$ fils et dont les feuilles ont toute une profondeur q a plus de $\lceil \frac{m}{2} \rceil^q$ nœuds et donc plus de $\lceil \frac{m}{2} \rceil^q$ entrées. On en déduit que la profondeur d'un arbre avec N entrées est inférieure à $\log_m N$, et que la complexité de la recherche est $O(\log_m N \log m) = O(\log N)$.

Cette complexité est apparemment indépendante du paramètre m. Ceci est en fait trompeur, parce que dans une implémentation pratique utilisant une mémoire à forte latence, l'accès à un nœud nécessite une lecture du nœud en entier, et donc un coût $O(m)$.

Question 2. (a) Tout comme dans un arbre binaire de recherche, on va essayer d'insérer la nouvelle entrée en bas de l'arbre. Cependant, au lieu de créer un nouveau nœud, on va plutôt essayer de l'ajouter à un nœud existant. On commence donc par rechercher le nœud feuille dans lequel il faut insérer la nouvelle entrée, puis on l'y insère.

Si le nœud ne contient pas trop d'étiquettes, c'est fini, l'arbre vérifie les invariants voulus.

Si le nœud contient alors trop d'étiquettes, il faut alors le diviser en deux nouveaux nœuds et remplacer donc l'ancien nœud par les deux nouveaux dans son parent, qui contiendra donc aussi une nouvelle étiquette. En pratique, dans ce cas, l'ancien nœud contient exactement m étiquettes $x_0, \ldots, x_{m-1}$. On sépare cette séquence d'étiquettes en deux moitiés et une étiquette séparatrice : on peut prendre $x_0, \ldots, x_{\lceil \frac{m}{2} \rceil - 2}$ pour la première moitié, $x_{\lceil \frac{m}{2} \rceil - 1}$ pour l'étiquette séparatrice, et $x_{\lceil \frac{m}{2} \rceil}, \ldots, x_{m-1}$ pour la seconde moitié. Les deux moitiés, de tailles $\lceil \frac{m}{2} \rceil - 1$ et $\lfloor \frac{m}{2} \rfloor$, peuvent alors être stockée dans deux nouveaux nœuds. Il faut alors remplacer l'ancien nœud par ces deux nouveaux nœuds, et utiliser le séparateur $x_{\lceil \frac{m}{2} \rceil - 1}$ comme nouvelle étiquette pour le parent. À son tour, le parent peut contenir trop d'étiquettes. Il faut alors répéter l'opération sur le parent, et ainsi de suite jusqu'à la racine. Si la racine

doit elle-même être divisée en deux, on crée une nouvelle racine qui ne contiendra qu'une étiquette (ainsi, la hauteur de l'arbre augmentera).

Pour résumer, on peut utiliser le pseudo-code suivant :

```
Fonction Insere_aux(x, v, nœud)
  Si nœud = Feuille
    Renvoyer NouveauSousArbre(x, v, Feuille)
  Sinon
    i = Fonction Cherche_premier_supérieur(x, nœud.nombre_fils, nœud.étiquette)
    r = Insere_aux(x, v, nœud.fils[i])
    Si r = NouveauSousArbre(x', v', t)
      insère x' à la position i   dans nœud.étiquettes
      insère v' à la position i   dans nœud.valeurs
      insère t  à la position i+1 dans nœud.fils
      nœud.nombre_fils += 1
      Si nœud.nombre_fils > m
        nœud' = AllouerNouveauNœud()
        x' = nœud.étiquettes[(m-1)/2]
        v' = nœud.valeurs[(m-1)/2]
        nœud'.étiquettes = nœud.étiquettes[(m+1)/2..m-1]
        nœud'.valeurs    = nœud.valeurs[(m+1)/2..m-1]
        nœud'.fils       = nœud.fils[(m+1)/2..m]
        tronquer nœud.étiquettes à (m-1)/2 éléments
        tronquer nœud.valeurs    à (m-1)/2 éléments
        tronquer nœud.fils       à (m+1)/2 éléments
        Renvoyer NouveauSousArbre(x', v', nœud')
      Sinon
        Renvoyer Ok

Fonction Insere(x, v, arbre)
  r = Insere_aux(x, v, arbre.racine)
  Si r = NouveauSousArbre(x', v', t)
    nœud' = AllouerNouveauNœud()
    nœud'.étiquettes = [x']
    nœud'.valeurs = [v']
    nœud'.fils = [arbre.racine, t]
    arbre.racine = nœud'
```

Dans le pire cas, il faut rajouter un nœud en $O(m)$ pour chaque nœud d'une feuille jusqu'à la racine. La complexité en pire cas est donc $O(m \log_m N)$. Bien sûr, en pratique, les ajouts de nouveaux nœuds sont rares. S'il n'y a pas de tel ajout, la complexité est $O(m + \log_m N)$.

(b) Si, par exemple, on insère la séquence d'entiers $1, \ldots, N$, alors on va remplir les nœuds de la frange droite de l'arbre, puis les diviser en deux. Après une telle division, le nouveau nœud de droite restera inchangé. Ainsi, à part la frange de droite, tous les nœuds de l'arbre ne seront qu'à moitié pleins, c'est-à-dire que, à une entrée près, ils seront au minimum de leur remplissage.

(c) **Pas de pseudo-code demandé pour cette question. Une simple explication détaillée suffira.**

Le problème de l'algorithme précédent, c'est qu'on ne cherche pas à remplir les nœuds voisins lorsqu'un nœud est plein. Une variante consiste, lorsqu'un nœud est trop plein, à essayer de transférer des fils et des étiquettes vers ses frères gauche et droit immédiats avant de décider de diviser le nœud en deux parties. Lors d'un tel transfert, il faut aussi modifier l'étiquette séparatrice dans le nœud parent.

La complexité en temps dans le pire cas reste la même : $O(m \log_m N)$.

Dans cette variante, lorsque l'on crée un B-arbre à partir d'une séquence croissante d'étiquettes, l'essentiel des nœuds de l'arbre est plein, sauf ceux de la frange droite, qui sont en cours de remplissage.

Question 3. Pas de pseudo-code demandé pour cette question. Une simple explication détaillée suffira.

Tout comme pour un arbre binaire de recherche, supprimer une entrée d'un B-arbre qui ne se trouve pas dans un nœud feuille est plus difficile. On commence donc par se ramener à ce cas : si une étiquette dans un arbre B ne se situe pas dans un nœud feuille, alors son prédécesseur immédiat (qui existe forcément) est nécessairement stocké dans un nœud feuille. Ainsi, si on se trouve dans cette situation, on commence par supprimer le prédécesseur immédiat de l'entrée que nous souhaitons supprimer, puis on remplace dans l'arbre l'entrée que nous voulons vraiment supprimer par celle que nous venons de supprimer dans un nœud feuille.

Il faut donc résoudre le problème de la suppression d'une entrée de l'arbre se situant dans un nœud feuille. On procède de manière duale de l'algorithme proposé dans la question 2.c : on commence par supprimer l'entrée de son nœud feuille. Si le nœud en question ne contient alors pas assez d'entrées (il en contient donc $\lceil \frac{m}{2} \rceil - 2$), alors on essaie d'en transférer depuis ses frères gauche et droits (puisque $m \geq 3$, tout nœud a nécessairement au moins un frère). Si un frère gauche ou droit est lui-même à la limite basse de son remplissage (c'est-à-dire qu'ils ont chacun $\lceil \frac{m}{2} \rceil - 1$ étiquettes), alors on peut fusionner un frère avec le nœud courant : en comptant le séparateur provenant du parent, on obtient un nouveau nœud avec $2 \lceil \frac{m}{2} \rceil - 2$ étiquettes, soit moins que la limite supérieure de $m - 1$ étiquettes. Bien sûr, cette fusion de nœuds peut poser un problème pour le parent, puisque celui-ci peut ne plus être assez rempli. Il faut donc continuer cette opération jusqu'à la racine. Arrivé à la racine, si celle-ci n'a plus qu'un fils à l'issue de l'opération, on la supprime et on prend son unique fils comme nouvelle racine.

Comme pour l'insertion, on effectue dans le pire cas $O(m)$ opérations à chaque niveau de l'arbre. La complexité dans le pire cas est donc $O(m \log_m N)$.

Tout comme pour l'insertion, dans la grande majorité des cas, il ne sera pas nécessaire de fusionner de nœuds, et la complexité sera alors $O(m + \log_m N)$.

Question 4. Tout d'abord, on se rend vite compte qu'il sera nécessaire d'extraire l'entrée la plus grande de T_1 ou la plus petite de T_2 pour servir de séparateur lors de la fusion. On peut faire cette opération grâce à l'algorithme de la question 3.

Le piège ensuite serait d'insérer l'un des deux arbres comme fils de la racine de l'autre, ou, pire, de créer un nouvelle racine comme parent des racines des deux arbres. En effet, une telle opération, dans le cas général, casserait l'invariant selon lequel toutes les feuilles ont la même hauteur.

On distingue plusieurs cas :

— Si les deux arbres ont la même hauteur, alors on essaie de fusionner leurs racines. Si la fusion des racines est trop grosse, alors on fait comme lors d'une insertion : on divise la racine en deux, et on crée une nouvelle racine.

— Si les deux arbres n'ont pas la même hauteur, supposons par exemple que T_1 soit plus haut. Dans ce cas, on cherche dans la frange droite de T_1 le nœud dont la hauteur est immédiatement supérieure à la hauteur de T_2, et on y insère T_2 comme dernier fils en utilisant le séparateur préalablement extrait. Il faut alors vérifier que ce nœud n'est pas trop plein, et réutiliser les algorithmes de la question 2 si nécessaire.

Question 5. Lors de l'insertion d'une nouvelle entrée, en utilisant la technique de la question 2.c., on peut se ramener au cas où on divise un nœud qui si ses (ou son) frères immédiats sont entièrement pleins. Dans ce cas, au lieu de diviser le nœud en deux, on peut plutôt choisir de le fusionner avec son

voisin plein et de diviser le tout en trois. Cela permet effectivement de respecter la borne inférieure $\left\lceil \frac{2m-1}{3} \right\rceil$.

Pour la suppression, on remarque la même chose : on peut se ramener au cas où il faut fusionner trois nœuds qui sont à la limite inférieure du remplissage en deux nœuds.

Néanmoins, si on utilise ces algorithmes, il faut relâcher la contrainte sur le nombre de fils de la racine, puisque celle-ci n'a pas de frère avec lequel on pourrait fusionner avant de diviser en trois. Ainsi, on doit autoriser la racine à avoir entre 2 et $2\left\lfloor \frac{2m-2}{3} \right\rfloor + 1$ fils (inclus).

Pour plus d'informations sur les B-arbres, on pourra consulter [Knu98, §6.2.4] ou [Wik21].

Références

[Knu98] Donald E. Knuth. *The Art of Computer Programming, Volume 3 : Sorting and Searching.* Addison Wesley, 1998.

[Wik21] Wikipedia. B-tree, 2021. `https://en.wikipedia.org/wiki/B-tree`.

Exercice 1. Suppression d'arête

1. Rappeler la définition d'une structure de union-find.

Soit T un graphe à n sommets obtenu à partir d'un arbre en ajoutant une arête.

2. Donner un algorithme (en pseudo-code) pour trouver une arête de T que l'on peut enlever de façon à obtenir un arbre. On essaiera d'avoir une complexité aussi faible que possible.
3. Implémenter l'algorithme précédent en OCaml. Le graphe est supposé être donné sous forme de liste d'adjacence.

Une arborescence est un graphe orienté acyclique ayant une racine r et tel que tout sommet autre que la racine possède un degré entrant égal à 1.
Soit $\vec{T}$ un graphe à n sommets obtenu à partir d'une arborescence en ajoutant un arc.

4. Reprendre les questions précédentes pour trouver un arc de $\vec{T}$ que l'on peut enlever de façon à obtenir une arborescence.

Exercice 2. Arbre de segments

Soit `a` un tableau d'entiers.
On souhaite concevoir, à partir de `a`, une structure de donnée `t` permettant de réaliser efficacement les opérations suivantes :

- `min_range i j t` : renvoie le minimum des éléments de `a` entre les indices `i` et `j`.
- `set i e t` : met à jour la structure pour que `a.(i)` soit remplacé par `e`.

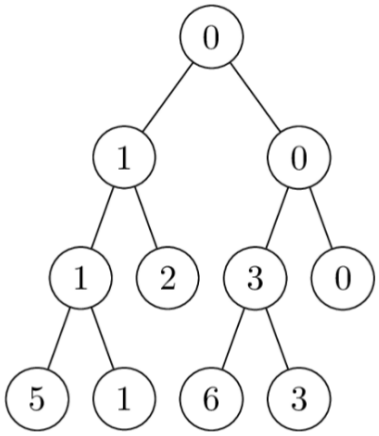
1. Proposer une solution naïve et donner sa complexité.

Dans la suite, on va utiliser un arbre de segments :

Définition : Arbre de segments

Un arbre de segments (pour un tableau `a`) est un arbre binaire dont :
- Les feuilles sont les éléments de `a`
- Chaque noeud est étiqueté par un triplet `(m, i, j)` tel que son sous-arbre contienne les feuilles `a.(i), ... , a.(j)` et `m` est le minimum de ces valeurs.

Par exemple, voici un arbre de segments obtenu à partir du tableau `[|5; 1; 2; 6; 3; 0|]`, où on a représenté seulement les minimums (premiers éléments de chaque triplet) :



Ainsi, les feuilles sont bien les éléments du tableau `[|5; 1; 2; 6; 3; 0|]` et chaque noeud correspond à un minimum sur une certaine plage du tableau.
Remarque : Il y a d'autres arbres de segments possibles pour le même tableau.
On utilisera le type suivant :

```
type tree = E | N of int * int * int * tree * tree
```

Ainsi, un sous-arbre `N(m, i, j, g, d)` possède `a.(i)`, `a.(i + 1)`, ..., `a.(j)` comme feuilles, de minimum `m` et de sous-arbres `g`, `d`.

2. Écrire une fonction `make : int array -> tree` qui construit un arbre de segments à partir d'un tableau d'entiers. On fera en sorte que l'arbre construit soit de hauteur logarithmique en la taille du tableau.
3. Écrire une fonction `set i e t` qui met à jour `t` en remplaçant `a.(i)` par `e`.
   Quelle est la complexité de cette fonction?
4. Écrire une fonction `min_range i j t` renvoyant le minimum des éléments de `a` entre les indices `i` et `j`.
5. Montrer que la complexité de `min_range i j t` est $O(\log(n))$, où n est la taille de `a`.
6. On s'intéresse à un autre problème : calculer efficacement une somme d'éléments entre les indices `i` et `j`, dans un tableau. Adapter les fonctions précédentes pour y parvenir.