**Nom :**
**Prénom :**
**Login EPITA :**
**UID :**

# Examen d'algorithmique

EPITA ING1 2010 S1, A. DURET-LUTZ

Durée : 1 heure 30

Janvier 2008

## Consignes

- Tous les documents (sur papier) sont autorisés.
  Les calculatrices, téléphones, PSP et autres engins électroniques ne le sont pas.
- Répondez sur le sujet dans les cadres prévus à cet effet.
- Il y a 6 pages. Rappelez votre UID en haut de chaque page au cas où elles se mélangeraient.
- Ne donnez pas trop de détails. Lorsqu'on vous demande des algorithmes, on se moque des points-virgules de fin de commande etc. Écrivez simplement et lisiblement. Des spécifications claires et implémentables sont préférées à du code C ou C++ verbeux.
- Le barème est indicatif et correspond à une note sur 20.

## File (4 points)

On souhaite représenter un ensemble dynamique $S$ d'éléments entiers. Cet ensemble, initialement vide, peut être augmenté ou diminué par les opérations suivantes :
- `insert` ajoute un élément à $S$, et
- `extract_min` retire de $S$ son plus petit élément.

1. **(1 point)** En une phrase, de quelle façon doit-on modifier l'implémentation par tas de la file de priorité vue en cours pour pouvoir effectuer ces deux opérations efficacement ? (Rappel : la file de priorité du cours supporte `insert` et `extract_max`.)

*Réponse :*

On inverse la comparaison qui sert à ordonner les éléments du tas lors de l'insertion et des réorganisations. Ainsi c'est toujours le plus petit élément qui sera en haut du tas et non le plus grand.
Notez que les deux opérations doivent être modifiées, pas seulement l'insertion.

2. **(1 point)** Avec cette nouvelle implémentation quelles sont les complexités du temps d'exécution de `insert` et `extract_min` en fonction du nombre $n$ d'éléments de $S$ ?

*Réponse :*

Comme effectuer un $<$ n'est pas plus coûteux qu'effectuer un $>$, les complexités de ces deux opérations sont les mêmes que celles de `insert` et `extract_max` dans l'implémentation d'origine, c'est-à-dire $O(\log n)$.

3. **(1 point)** Toujours avec cette implémentation par tas, donnez l'état du tableau représentant $S$ après y avoir effectué chacune des opérations suivantes :
   - `insert(6)`

   | 6 |
   |---|

- `insert(2)`

| 2 | 6 |
|---|---|

- `insert(5)`

| 2 | 6 | 5 |
|---|---|---|

- `insert(3)`

| 2 | 3 | 5 | 6 |
|---|---|---|---|

- `extract_min()`

| 3 | 6 | 5 |
|---|---|---|

- `insert(7)`

| 3 | 6 | 5 | 7 |
|---|---|---|---|

- `insert(8)`

| 3 | 6 | 5 | 7 | 8 |
|---|---|---|---|---|

- `extract_min()`

| 5 | 6 | 8 | 7 |
|---|---|---|---|

4. **(1 point)** Sur cette structure de données, quelle serait la complexité d'une opération `extract_next_to_min` pour retirer de $S$ le deuxième plus petit élément (sans retirer le premier) ? Justifiez votre réponse.

*Réponse :*

$O(\log n)$ avec plusieurs justifications possibles :
- il suffit de faire `a=extract_min(); b=extract_min(); insert(a); return b;` c'est-à-dire trois opérations en $O(\log n)$.
- le second plus petit élément du tas est soit le fils gauche de la racine, soit le fils droit : on peut déterminer lequel en une comparaison ($O(1)$), puis il suffit de faire l'extraction à partir de ce fils selon la même méthode qu'une extraction à partir de la racine ($O(\log n)$).

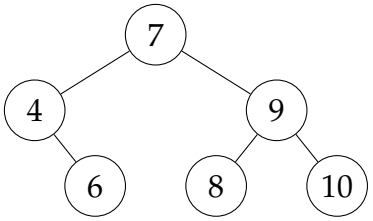
# Arboriculture (6 points)

Dans toutes ces questions, on considère des arbres binaires d'entiers.

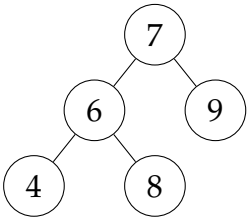
1. **(2 points)** Proposez un algorithme récursif retournant `true` si et seulement si l'arbre binaire passé en argument est un Arbre Binaire de Recherche.

   Attention, la première idée n'est pas toujours la bonne. Assurez-vous au moins que votre algorithme fonctionne sur les deux exemples qui suivent.

Cet arbre est un ABR :




Celui-là n'en est pas un :



*Réponse :*

Il s'agit de s'assurer que chaque nœud est plus grand que tous les nœuds de son sous-arbre gauche, et plus petit que tous les nœuds de son sous-arbre droit.

Plusieurs approche sont possibles.

Une première consiste à vérifier cette contrainte de bas en haut. Pour cela on effectue un parcours récursif pendant lequel on calcule le maximum et le minimum de chaque sous arbre. Les trois valeurs retournée par check_abr indique si les sous-arbre de racine nœud est un ABR ainsi que ses valeurs minimales et maximales.

```
is_abr(nœud) ↦ bool
    (b, min, max) ← check_abr(nœud)
    return b
check_abr(nœud) ↦ (bool, int, int)
    if (nœud = nil) return (⊤, −∞, +∞)
    (gabr, gmin, gmax) ← check_abr(left_child(nœud))
    (dabr, dmin, dmax) ← check_abr(right_child(nœud))
    if (gabr && dabr && gmax ≤ key(nœud) && dmin ≥ key(nœud))
        return (⊤, gmin, dmax)
    else
        return (⊥, gmin, dmax)
```

Une seconde consiste à vérifier cette contrainte de haut en bas. Au fûr et à mesure que l'ont descent dans les sous-abres, on restreint la plage des valeurs acceptées en fonction des clefs que l'on a recontrées. C'est beaucoup plus élégant.

```
is_abr(nœud) ↦ bool
    return check_abr(nœud, −∞, +∞)

check_abr(nœud, min, max) ↦ bool
    if (nœud = nil) return ⊤
    if (key(nœud < min or key(nœud) > max) return ⊥
    return check_abr(left_child(nœud, min, key(nœud))) et
                                        check_abr(right_child(nœud, key(nœud), max))
```

Une autre approche serait de vérifier que les valeurs sont bien ordonnées dans l'ordre infixe. Cela peut se faire dans un parcours en profondeur qui utilise une variable globale pour indiquer la dernière valeur lue dans l'ordre infixe.

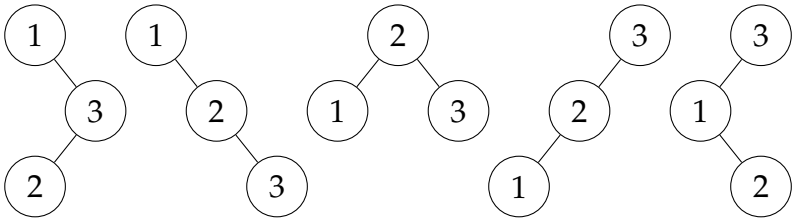
Vous avez été très nombreux à proposer des algorithmes qui ne vérifiaient que l'ordre entre un nœud est ses deux fils immédiats. Avec un tel test, votre algorithme pense que l'arbre donné en 2e exemple est un ABR. Certains d'entre vous ont généralisé l'algorithme pour faire des comparaisons additionnelles avec la racine de l'arbre ou avec le père du nœud courant : dans les deux cas il est possible de trouver des exemples d'arbres qui seront reconnus par erreur comme des ABR.

2. **(1 point)** Donnez la complexité de cet algorithme en fonction du nombre $n$ de nœuds de l'arbre inspecté. Justifiez votre réponse.

*Réponse :*

$O(n)$ car au pire l'algorithme effectue un nombre constant d'opération sur chacun des nœuds de l'arbre. (Il peut s'arrêter plus tôt s'il trouve que l'arbre n'est pas un ABR.)

3. **(2 points)** Notons $A(n)$ le nombre d'arbres binaires de recherche différents qui permettent de représenter $n$ entiers donnés. Par exemple il existe 5 ABR représentant $\{1, 2, 3\}$ :




On a $A(0) = 1$ (l'arbre vide), $A(1) = 1$, $A(2) = 2$, $A(3) = 5$, $A(4) = 14$.
Donnez une définition récursive de $A(n)$.

*Réponse :*

(L'énoncé du partiel était erroné : il indiquait $A(3) = 3$ et $A(4) = 10$. J'ai offert un point de bonus aux 14 étudiants qui l'ont signalé, qu'ils aient répondu à la question ou non.)

Supposons que l'on connaisse $A(1), \ldots, A(n-1)$. On cherche à exprimer $A(n)$
Considérons $n$ entiers ordonnés. Combien d'ABR permettent de représenter cet ensemble de $n$ entiers en utilisant le $k^{\text{e}}$ entier pour racine ?
Il y a $A(k-1)$ possibilités pour le sous-arbre gauche, et $A(n-k)$ possibilités pour le sous-arbre droit. Soit $A(k-1)A(n-k)$ ABR ayant $k$ pour racine.
Maintenant envisageons toutes les racines possibles, on a :

$$\begin{cases} A(0) = 1 \\ A(n) = \sum_{k=1}^{n} A(k-1)A(n-k) = \sum_{k=0}^{n-1} A(k)A(n-1-k) \end{cases}$$

Ce nombre est le $(n-1)^{\text{e}}$ nombre de Catalan. Nous l'avons recontré en cours lorsque nous avons compté toute les façons de parenthéser un produit de matrices. On sait aussi que $A(n) = \frac{1}{n}C_{2n-2}^{n-1} = C_{2n-2}^{n-1} - C_{2n-2}^{n-2}$ mais seules l'une des sommes ci-dessus était attendue.

4. **(1 point)** Notons $B(n)$ le nombre d'arbres binaires différents que l'on peut construire en utilisant $n$ nœuds. (Cette fois-ci les arbres ne sont pas des arbres de recherche.) On a $B(3) = 5$ car il n'existe que cinq « formes » d'arbres qui utilisent 3 nœuds.
   Donnez une définition de $B(n)$. (Vous pouvez y utiliser $A(n)$.)

*Réponse :*

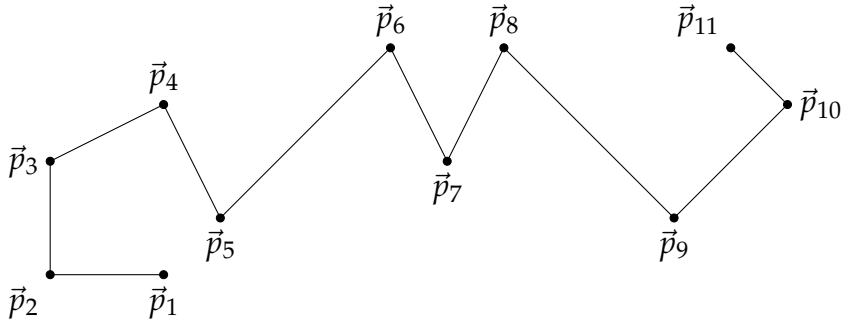
$$B(n) = A(n)$$

Prennez $n$ nœuds est construisez un arbre binaire quelconque. Effectuez un parcours infixe pour numéroter les nœuds de 1 à $n$ : vous obtenez un arbre binaire de recherche. Pour compter le nombre de formes d'arbres binaires que l'on peut construire avec $n$ nœuds il suffit donc de compter le nombre d'arbres binaires de recherche qui peuvent représenter les entiers $1, \ldots, n$.

# Segmentation par les moindres carrés (6 points)

Dans cet exercice nous souhaitons épurer un chemin enregistré par un récepteur GPS. Le récepteur GPS enregistre sa position à intervalles réguliers, ce qui nous permet de tracer le chemin parcouru en reliant ces positions. Pour simplifier nous allons travailler en deux dimensions, sur un plan.

Voici un exemple de chemin possédant 11 points notés $\vec{p}_1 \ldots \vec{p}_{11}$. Dans le cas général on notera $n$ le nombre de points enregistrés.

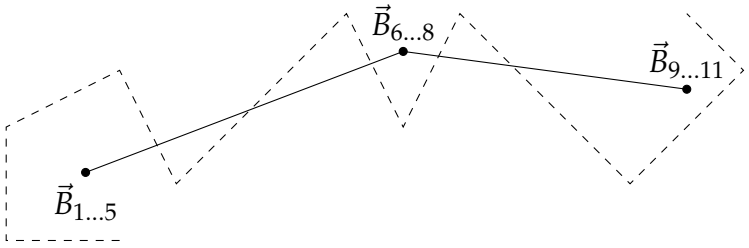



Notre objectif est de simplifier cette ligne brisée pour obtenir un nombre de points $G \leq n$ donné. On va pour cela regrouper les $n$ points en $G$ groupes de points consécutifs ; chaque groupe de points sera représenté par son barycentre. Les groupes seront formés de façon à minimiser la somme des carrés des distances entre chacun des points $\vec{p}_k$ et le barycentre $\vec{B}_{i\ldots j}$ de leur groupe.

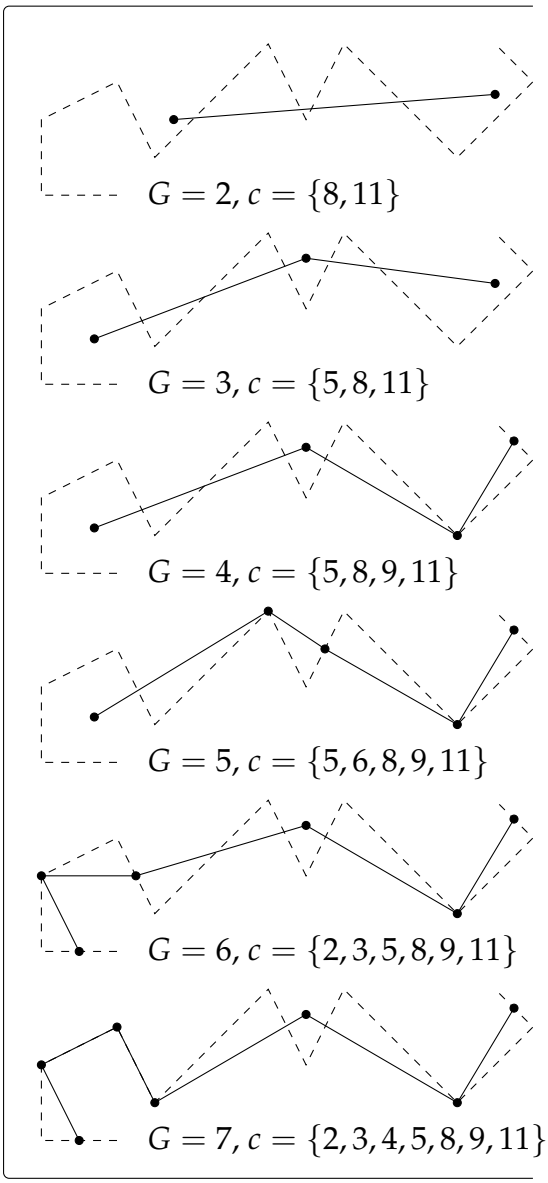
Par exemple pour $G = 3$, un regroupement optimal des points de notre exemple est

$$\underbrace{\vec{p}_1, \vec{p}_2, \vec{p}_3, \vec{p}_4, \vec{p}_5}_{\vec{B}_{1\ldots5}}, \underbrace{\vec{p}_6, \vec{p}_7, \vec{p}_8}_{\vec{B}_{6\ldots8}}, \underbrace{\vec{p}_9, \vec{p}_{10}, \vec{p}_{11}}_{\vec{B}_{9\ldots11}}$$

Graphiquement, on a :




Comme on ne regroupe que des points consécutifs, on pourra représenter un regroupement par l'ensemble $c$ des indices des derniers points de chaque groupe. Dans notre exemple avec $G = 3$, on a $c = \{5, 8, 11\}$. L'encart de droite montre des regroupements optimaux pour diverses valeurs de $G$ sur ce même exemple.




Formalisons tout ceci plus proprement. Tout d'abord, définissons $\vec{B}_{i..j}$ le barycentre des vecteurs $\vec{p}_i$ à $\vec{p}_j$ et $\mathrm{SDC}(i, j)$ la somme des distances au carré entre ces vecteurs et leur barycentre :

$$\vec{B}_{i\ldots j} = \frac{1}{j-i+1} \sum_{k=i}^{j} \vec{p}_k$$

$$\mathrm{SDC}(i, j) = \sum_{k=i}^{j} \left\| \vec{p}_k - \vec{B}_{i\ldots j} \right\|^2$$

Pour un $G$ et un ensemble de $\vec{p}_i$ donnés, notre objectif est de trouver l'ensemble $c = \{c_1, c_2, \ldots, c_G\}$ qui minimise $D = \mathrm{SDC}(0, c_1) + \mathrm{SDC}(c_1 + 1, c_2) + \cdots + \mathrm{SDC}(c_{G-1} + 1, c_G)$.

On cherche à minimiser cette distance par programmation dynamique.

Notons $P(k, j)$ la valeur de $D$ minimale que l'on peut obtenir pour le problème réduit à la recherche de $k$ groupes parmi les $j$ premiers points. On a, pour tout $j \leq n$ :

$$P(1, j) = \text{SDC}(1, j)$$
$$P(k, j) = \min_{i \in \{k-1, \ldots, j-1\}} \{P(k-1, i) + \text{SDC}(i+1, j)\} \quad \text{si } 2 \leq k \leq G$$

1. **(2 points)** Justifiez brièvement cette dernière définition récursive.

*Réponse :*

On fait les regroupement en partant des derniers points.
Pour choisir $k$ groupes il suffit de former un premier groupe entre $i+1$ et $j$ puis de former $k-1$ groupes entre 1 et $i$. Reste à choisir de $i$ qui minimise $SDC(i+1, j)$ (le poids du premier groupe) et $P(k-1, i)$ (le poids des $k-1$ autres groupes).
Les bornes de recherche pour $i$ s'expliquent par le fait que si l'on cherche le $i$ pour un $k$ donné, il reste encore $k-1$ découpages à faire, donc $i$ ne peut pas être dans les $k-1$ premières valeurs.

2. **(2 points)** Sachant qu'une astuce de calcul (non décrite ici) permet de calculer $\text{SDC}(i, j)$ en $\Theta(1)$, quelle serait, en fonction de $G$ et $n$, la complexité d'un algorithme calculant $P(G, n)$ par programmation dynamique ?

*Réponse :*

La définition de $P$ implique de remplir un tableau de taille $G \times n$. Pour chacune des cases, le calcul de $P(k, j)$ demande $\text{O}(n)$ opérations (la boucle pour le min dans laquelle l'accès au valeurs de $P(k-1, i)$ et $SDC(i+1, j)$ se font en temps constant).
Le calcul de $P(G, n)$ est donc en $G \times n \times \text{O}(n) = \text{O}(G \times n^2)$.

3. **(2 points)** Une fois que l'on possède un algorithme de programmation dynamique pour calculer $P(k, j)$, comment le modifier pour obtenir $c_1, c_2, \ldots, c_G$ ?

*Réponse :*

Il faut retenir les $i$ qui ont été choisis pour le calcul de chaque $P(k, j)$.
On a $c_G = n$.
$c_{G-1}$ correspond au $i$ choisi pour $P(G, c_G)$.
$c_{G-2}$ correspond au $i$ choisi pour $P(G-1, c_{G-1})$.
etc.

# Recherche ternaire (4 points)

L'algorithme de recherche ternaire dans un tableau trié fonctionne comme celui de la recherche binaire (ou dichotomique) mais divise le problème en trois plutôt qu'en deux.

1. **(2 points)** Proposez un algorithme récursif de recherche ternaire.

*Réponse :*

search3($A, val, debut, fin$)$\mapsto$ int
    if ($fin < debut$) return $-1$
    $pt_1 \leftarrow debut + \lfloor \frac{fin-debut}{3} \rfloor$
    $pt_2 \leftarrow debut + \lfloor 2\frac{fin-debut}{3} \rfloor$
    if ($val < A[pt_1]$) return search3($A, val, debut, pt_1 - 1$)
    if ($A[pt_1] = val$) return $pt_1$
    if ($val < A[pt_2]$) return search3($A, val, pt_1 + 1, pt_2 - 1$)
    if ($A[pt_2] = val$) return $pt_2$
    return search3($A, val, pt_2 + 1, fin$)

Une erreur fréquente était d'écrire $pt_1 \leftarrow \lfloor \frac{debut+fin}{3} \rfloor$ en s'inspirant d'un calcul de moyenne : il suffit de faire les calculs avec par exemple $debut = 10$ et $fin = 15$ pour voir que $pt_1 = 8$ tombe en dehors des bornes.

2. **(1 point)** Calculez et justifiez la complexité de cet algorithme en fonction de la taille $n$ du tableau.

*Réponse :*

Notons $T(n)$ la complexité de search3().
Chaque exécution de search3() fait :
- un nombre constant d'opérations si elle trouve la valeur en $pt_1$ ou $pt_2$, ou si $fin < debut$.
- un nombre constant d'opérations plus $T(n/3)$ opérations si elle ne trouve pas la valeur et doit s'appeler récursivement sur un tiers du tableau.

On a donc $T(n) \leq T(n/3) + \Theta(1)$.
Le théorème général nous donne $T(n) = \mathrm{O}(\log_3 n) = \mathrm{O}(\log n)$.

En réalité l'appel récursif ne reçoit pas exactement $n/3$ valeurs. À cause des parties entières et de l'exclusion de $pt_1$ et $pt_2$ il peut en recevoir un peu moins. On s'en fiche. D'une part ce détail est négligeable quand $n$ tends vers $\infty$, d'autre part mous majorons $T(n)$.

3. **(1 point)** Est-il plus intéressant de faire une recherche binaire ou ternaire ?

*Réponse :*

La question est assez vague car elle ne précise pas ce qui est entendu par « intéressant ». J'attendais tout de même une réponse justifiée.
Asymptotiquement, les deux recherches ont la même complexité : $\mathrm{O}(\log n)$. Choisir l'une ou l'autre au sein d'un plus gros algorithme n'aura donc aucune influence sur la complexité de cet algorithme.
Cette réponse me suffisait, mais on peut ensuite discuter d'autres aspects. Par exemple la recherche binaire est plus courte à écrire, donc plus simple à vérifier et probablement plus simple à optimiser.
On peut essayer de comparer plus précisément les complexités. Si on note $m$ le coût du calcul d'un point moyen (c'est-à-dire $pt_1$ ou $pt_2$ ci-dessus), $c$ le coût d'une comparaison $<$ avec son if, $e$ le coût d'un test d'égalité avec son if, et $a$ le coût de l'appel récursif (tous les appels récursifs sont terminaux, donc ce coût est faible) on peut dire que la recherche ternaire coûte au pire $\log_3(n)(2m + 3c + 2e + a)$ alors que la recherche binaire coûte au pire $\log_2(n)(m + 2c + e + a)$. La recherche ternaire sera plus lente si le ratio de ces deux fonctions est supérieur à 1 :

$$\frac{\frac{\log_2(n)}{\log_2(3)}(2m + 3c + 2e + a)}{\log_2(n)(m + 2c + e + a)} > 1 \iff \frac{2m + 3c + 2e + a}{m + 2c + e + a} > \log_2(3)$$

On voit que la préférence d'une méthode à l'autre dépend uniquement des valeurs de $m$, $c$, $e$ et $a$, c'est-à-dire de la machine, et non de la taille du tableau dans lequel on effectue la cherche.