Université de Savoie, UFR SFA
INFO626, L3 STIC ISC

11 avril 2017
Durée : 1h30

Examen, INFO626

Documents autorisés : tous documents du cours/td/tp, notes manuscrites (nb : pas de livres)

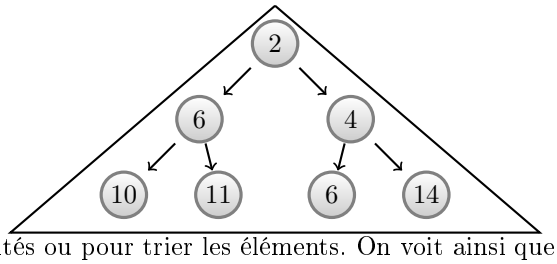
Les exercices sont indépendants. Le barème est indicatif. Il dépasse volontairement 20 pour que vous ayez le choix dans les exercices.

Exercice 1. Tas (/10)

On rappelle qu'un *tas* est un tableau dans les éléments sont organisés de manière spécifique, qui imite un arbre binaire. D'une part, le plus petit élément est à l'indice 1 du tableau et est appelé *racine* (l'indice 0 n'est pas utilisé). D'autre part, un élément à l'indice i du tableau est plus petit que l'élément à l'indice $2i$ et que l'élément à l'indice $2i+1$, s'ils existent. On dit que l'élément à l'indice $2i$ est le *fils gauche* de l'élément à l'indice i, et que l'élément à l'indice $2i+1$ est le *fils droit* de l'élément à l'indice i. Enfin, l'élément à l'indice i est le *père* des éléments aux indices $2i$ et $2i+1$. Notons que la dernière ligne peut être incomplète à droite, si le nombre d'éléments n n'est pas de la forme $2^k - 1$.

Par exemple, le tas ci-dessous stocké sous forme de tableau ainsi à gauche, correspond à la structure hiérarchique à droite :

indices	1	2	3	4	5	6	7	...
valeurs	2	6	4	10	11	6	14	...



Les tas sont très utiles pour faire des files à priorités ou pour trier les éléments. On voit ainsi que le plus petit élément est toujours à l'indice 1. Pour faire un tri, il suffit de sortir cet élément, de mettre le dernier élément du tas à la place, puis de le pousser dans une branche jusqu'à ce que la propriété de croissance soit respectée partout.

1. Si $n = 2^k - 1$ est le nombre d'éléments, combien y a-t-il de lignes dans l'arbre représentant le tas ? Combien y a-t-il de valeurs sur la dernière ligne ?

   Il y a k lignes dans l'arbre, et $2^{k-1} = \lceil \frac{n}{2} \rceil$ sur la dernière ligne.

2. On s'intéresse à la création du tas lorsque l'on insère un nouvel élément dedans (ici des entiers). On peut écrire la procédure INSERER ainsi :

```
   // Insère un entier dans une position valide.
1  Action INSERER( ES T : Tas, E e : entier);
   Var : i : entier;
2  début
3  |  T.dernier ←T.dernier+1;
4  |  T.elems[ T.dernier ] ←e;
5  |  i ←T.dernier /* i est la position courante de e */ ;
6  |  Tant Que i > 1 et T.elems[i] < T.elems[ i/2 ] Faire
   |  |  // On le remonte dans le tas en l'échangeant avec son père.
7  |  |  Echange( T.elems[i], T.elems[ i/2 ] );
8  |  |  i ←i / 2;
9  fin
```

Quel est la complexité en pire cas d'une insertion d'un nouvel élément dans un tas qui a déjà m éléments ? Justifiez brièvement.

On insère en dernière position puis on remonte l'élément ligne par ligne au maximum jusqu'à la racine. L'élément ajouté est sur la dernière ligne. Comme il y avait m éléments, l'élément $m+1$-ième est sur la $1+\log_2(m+1)$-ième ligne. Au pire, INSERER a un coût proportionnel à ce nombre donc $\in O(\log m)$.

3. En déduire la complexité en pire cas de la création d'un tas à $n = 2^k - 1$ éléments au total, obtenu en insérant progressivement ces n éléments par n appel de InsererTas.

   On va insérer n éléments, chacun à un coût $O(\log m)$, où m est le nombre de lignes du tas en construction. On simplifie en constatant que $\log m \leq \log n$. La complexité totale est donc en $O(n \log n)$.

4. On va utiliser une autre stratégie pour l'insertion, qui ne peut marcher que si on connait *a priori* le nombre d'éléments que l'on veut insérer. On l'écrit ainsi avec les actions ConstruitTas et InsèreFin.

```
   // Crée un tas T formé des n éléments du tableau P en les insérant par la fin
1  Action ConstruitTas( S T : Tas, E P[1..n] : Tableau d'entiers, E n : entier);
   Var : j :entier;
2  début
3  |  CréerTasVide(T,n) /* Crée un tas vide avec assez d'espace mémoire pour n éléments */
   |  Pour j de n à 1 par pas de −1 Faire
4  |  |  InsèreFin(T, P[j], j, n);
5  fin
```

```
   // Insère v dans le tas T en position i, en supposant que les éléments entre i+1
      et n sont déjà insérés.
1  Action InsèreFin( ES T : Tas, E v : entier, E i : entier, E n : entier );
2  début
3  |  T.elems[i] ←v ;
4  |  Tant Que 2 ∗ i + 1 <= n et v > min(T.elems[2 ∗ i], T.elems[ 2 ∗ i + 1 ]) Faire
5  |  |  si v > T.elems[2 ∗ i] alors j ←2 ∗ i;
6  |  |  sinon j ←2 ∗ i + 1;
7  |  |  Echange( T.elems[i], T.elems[j] );
8  |  |  i ←j;
9  fin
```

5. On suppose que l'on appelle ConstruitTas sur le tableau $P[] = \{14, 7, 12, 3, 0, 20, 9\}$. Que vaut le tas après construction ? (i.e. les quelles sont les valeurs dans le tableau T.elems ?)

   0 3 9 14 7 20 12

6. Dans la suite $n = 2^k - 1$. Quelle est la complexité en pire cas d'un appel à InsèreFin ?

   La complexité dépend du nombre de lignes en bas du tas, donc au pire $\log n$.

7. Néanmoins, quelle est la complexité en pire cas d'un appel à InsèreFin(T,v,i,n) lorsque $\frac{n}{2} < i \leq n$?

   En temps constant, car il n'y a pas d'échange sur la dernière ligne. Donc en $O(1)$.

8. De même, quelle est la complexité en pire cas d'un appel à InsèreFin(T,v,i,n) lorsque $\frac{n}{4} < i \leq \frac{n}{2}$? Et plus généralement lorsque $\frac{n}{2^j} < i \leq \frac{n}{2^{j-1}}$?

   On voit qu'il y a 2 lignes lorsque $\frac{n}{4} < i \leq \frac{n}{2}$, 3 lignes lorsque$\frac{n}{8} < i \leq \frac{n}{4}$, et plus généralement j lignes $\frac{n}{2^j} < i \leq \frac{n}{2^{j-1}}$. Donc la complexité est en $O(j)$.

9. Si maintenant vous sommez les complexités précédentes pour tous les éléments, vous obtiendrez la complexité amortie totale des n appels à InsèreFin, donc de ConstruitTas. Quelle est-elle ? Quelle est alors la complexité amortie de chaque appel à InsèreFin ? Fait-on mieux que la première méthode ?

On somme les complexités par groupe, en les multipliant par le nombre d'éléments dans le groupe.

$$\begin{aligned} T(n) &= \frac{n}{2} \times 1 + \frac{n}{2^2} \times 2 + \frac{n}{2^3} \times 3 + \ldots + \frac{n}{2^j} \times j + \ldots + 1 \times \log n \\ &= n \times \left(\frac{1}{2} \times 1 + \frac{1}{2^2} \times 2 + \frac{1}{2^3} \times 3 + \ldots + \frac{1}{2^j} \times j + \ldots \right) \\ &\leq 2 \times n. \end{aligned}$$

On est donc en temps linéaire, et on fait mieux que la 1ère méthode.

NB : Faites la somme des complexités par groupes (d'abord les éléments de n à $\frac{n}{2}$, puis ceux de $\frac{n}{2}$ à $\frac{n}{4}$, etc).
NB2 : Utilisez le fait que $\frac{1}{2} + \frac{2}{2^2} + \frac{3}{2^3} + \frac{4}{2^4} + \cdots \leq 2$.

Exercice 2. Notations O, Ω, Θ (/5)

Complétez le tableau ci-dessous en indiquant si f est un grand O, un grand Ω ou un grand Θ de la fonction g (pour n tendant vers l'infini). Attention, Juste $= +0,5$, Faux $= -0,5$, Rien $= +0$, ceci pour éviter que vous ne répondiez au hasard.

		g			
		$n \log n$	$3n^2 + 10n$	$n2^n$	$n\sqrt{n}$
f	$3n$	O	O	O	O
	$2n^2 + \log n$	Ω	Θ	O	Ω
	$\frac{1}{4} n \log n$	Θ	O	O	O

Exercice 3. Polygones simples, convexité et points intérieurs (/10)

Si $Q = (q_i)_{i=0..n-1}$ est un polygone simple, il existe une façon relativement simple de savoir si un point p est à l'intérieur de Q. On trace un rayon $[p, r)$ à partir de p, où r est un point différent de p, et on compte le nombre de fois où $[p, r)$ intersecte le bord de Q. Si ce nombre est impair alors le point est à l'intérieur, sinon le point est à l'extérieur (voir illustration page suivante).

Nb : dans la suite, pour simplifier, on supposera toujours que le rayon $[p, r)$ ne traverse pas un sommet de Q. C'est une hypothèse réaliste si r est bien choisi ou si r est tiré au hasard.

Nb : on vous redonne dans la page après quelques fonctions de base de géométrie algorithmique, que vous pouvez utiliser dans vos algorithmes.

1. Adaptez la fonction INTERSECTION-SEGMENTS du cours (rappelée plus loin) pour en faire une fonction INTERSECTION-RAYON(p_1, p_2, p_3, p_4) qui retourne vrai si et seulement le rayon $[p_1, p_2)$ intersecte le segment ferme $[p_3, p_4]$.

```
bool intersection_rayon( Point p, Point r, Point p3, Point p4 )
{
  double dp = orientation( p3, p4, p );
  double d3 = orientation( p, r, p3 );
  double d4 = orientation( p, r, p4 );
  if ( ( d3 == 0.0 ) && ( d4 == 0.0 ) )
    return sur_segment( p, p3, p4 ) || sur_segment( p, p4, p3 );
  return ( ( ( d3 <= 0.0 ) && ( d4 >= 0.0 ) && ( dp >= 0.0 ) )
           || ( ( d3 >= 0.0 ) && ( d4 <= 0.0 ) && ( dp <= 0.0 ) ) );
}
```

2. Ecrire maintenant l'algorithme qui teste si un point p est à l'intérieur de Q. On attend ici un algorithme simple de complexité linéaire en le nombre de sommets de Q. Son prototype sera
Fonction ESTINTÉRIEUR ?*(**E** p : Point, **E** Q : Polygone) : booléen*
NB : On écrira $Q.n$ pour avoir le nombre de sommets de Q, et $Q[i]$ pour accéder au i-ème sommet, avec l'indice i pris modulo $Q.n$. N'oubliez pas de choisir un r.

```
bool est_interieur( Point p, const Polygon& Q )
{
  Point r = 0.5*( Q[Q.n()-1] + Q[0] );
  int nb = 0;
  for ( int i = 0; i < Q.n(); ++i )
    if ( intersection_rayon( p, r, Q[ i ], Q[ i + 1 ] ) )
      ++nb;
  return ( nb & 1 ) != 0;
}
```

3. Si maintenant Q est un polygone convexe, combien de fois le rayon $[p, r)$ peut-il intersecter le bord de Q ?

   Il y a trois possibilités : 0, 1 ou 2 fois. 1 correspond à un point intérieur.
   Nb (non demandé) : Un argument pour prouver cela est que les côtés tournent et forment des angles croissants. Lorsqu'un premier côté croise le rayon il le traverse dans un sens, le deuxième côté croisé le traverse dans l'autre sens. Si un troisième côté le traversait, il faudrait qu'il le traverse dans le sens du premier et ça vaudrait dire que le polygone n'est pas convexe (soit le polygone ferait plusieurs tours sur lui-même et ne serait pas simple, soit le polygone tournerait à gauche puis à droite et ne serait pas convexe).

4. En fait, on peut être beaucoup plus efficace dans le cas où Q est convexe (le polygone est supposé stocké dans l'ordre trigonométrique). Imaginez que l'on regarde la ligne polygonale allant de q_i à q_j sur le bord du polygone. Si q_i est d'un côté du rayon $[p, r)$ et q_j de l'autre côté, alors on peut trouver le côté potentiel d'intersection avec la droite (pr) par dichotomie. Ecrivez cet algorithme (efficace) CHERCHE-CÔTÉ, qui retourne l'entier k tel que la droite (pr) intersecte le côté $[q_k q_{k+1}[$.
   Fonction CHERCHE-CÔTÉ*(**E** p, r : Point, **E** Q : Polygone, **E** i,j : entier) : entier*

```
int cherche_cote( Point p, Point r, const Polygon& Q, int i, int j )
{
  if ( i+1 >= j ) return i;
  int m = (i+j)/2;
  double di = orientation( p, r, Q[i] );
  double dm = orientation( p, r, Q[m] );
  double dj = orientation( p, r, Q[j] );
  if ( di*dm > 0.0 ) return cherche_cote( p, r, Q, m, j );
  else               return cherche_cote( p, r, Q, i, m );
}
```

5. Quelle est la complexité en pire case de CHERCHE-CÔTÉ en fonction de i et j ? Puis en fonction en n ?

   Cette fonction est dichotomique et divise par 2 l'espace de recherche à chaque tour. Au début $k = j - i$ est la distance séparant les deux points. A chaque fois, on restreint l'intervalle de moitié. Il est bien connu (e.g. killer theorem, cas 2) que le nombre d'appel récursif est donc de $\log_2(k) = \log_2(j - i)$ et qui correspond à la complexité de la fonction. Comme $n \geq j - i$, on obtient une complexité en $O(\log_2 n)$.

6. Est-ce que le côté retourné par CHERCHE-CÔTÉ a toujours une intersection non vide avec $[p, r)$?

   Non. La droite intersecte le côté mais le rayon peut en fait partir de l'autre côté.

7. Si on choisit r comme étant un point milieu sur un côté de Q, quelles sont les possibilités d'intersection entre $[p, r)$ et Q ? Faites un dessin pour (les) illustrer. Quel est l'intérêt d'un tel choix ?

   $[p, r)$ intersecte toujours le côté qui contient r. Si p est extérieur à Q, ce rayon intersecte aussi (au maximum) un autre côté de Q, à peu près en face. De plus, par ce choix, on est sûr que les deux extrémités du côté choisi sont de par et d'autre de (p, r). On pourra donc appeler CHERCHE-CÔTÉ à partir de ces deux points.

8. On définit donc r comme étant le milieu entre q_{n-1} et q_n. En déduire l'algorithme rapide ESTINTÉRIEURCONVEXE ? qui teste si un point p est intérieur à un convexe Q. Quelle est sa complexité en pire cas en fonction de n ?

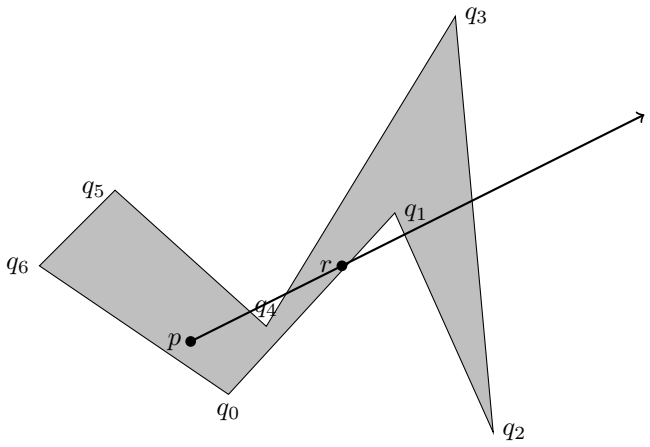


FIGURE 1 – Le point p est dans le polygone Q car $[p, r)$ intersecte 5 fois le bord de Q.

```
bool est_interieur_convexe( Point p, const Polygon& C )
{
  Point r = 0.5*(C[C.n()-1] + C[0]);
  int k = cherche_cote( p, r, C, 0, C.n() - 1 );
  int nb = 1; // [pr) croise déjà le côté [q_n-1,q_0]
  if ( intersection_rayon( p, r, C[ k ], C[ k + 1 ] ) )
    ++nb;
  return nb == 1;
}
```

```
   // Retourne un nombre > 0 si et seulement si r est à gauche du vecteur p⃗q, un
      nombre < 0 ssi r est à droite de p⃗q, et 0 si p, q, r sont alignés.
 1 Fonction ORIENTATION( E p, q, r : Point ) : réel ;
 2 début
 3 |  Retourner (q.x − p.x) ∗ (r.y − p.y) − (q.y − p.y) ∗ (r.x − p.x) ;
 4 fin
   // Sachant que r est sur la droite (pq), détermine si r appartient au segment [pq].
 5 Fonction SUR-SEGMENT( E p, q, r : Point ) : booléen ;
 6 début
 7 |  Retourner min(p.x, q.x) ≤ r.x ≤ max(p.x, q.x) et min(p.y, q.y) ≤ r.y ≤ max(p.y, q.y) ;
 8 fin
   // Détermine si les segments [p1, p2] et [p3, p4] s'intersectent.
 9 Fonction INTERSECTION-SEGMENTS( E p1, p2, p3, p4 : Point ) : booléen;
10 début
11 |  d1 ← ORIENTATION(p3, p4, p1);
12 |  d2 ← ORIENTATION(p3, p4, p2);
13 |  d3 ← ORIENTATION(p1, p2, p3);
14 |  d4 ← ORIENTATION(p1, p2, p4);
15 |  si ((d1 < 0 et d2 > 0) ou (d1 > 0 et d2 < 0)) et ((d3 < 0 et d4 > 0) ou (d3 > 0 et d4 < 0))
   |  alors Retourner Vrai ;
16 |  sinon si d1 = 0 et SUR-SEGMENT(p3, p4, p1) alors
17 |  |  Retourner Vrai ;
18 |  sinon si d2 = 0 et SUR-SEGMENT(p3, p4, p2) alors
19 |  |  Retourner Vrai ;
20 |  sinon si d3 = 0 et SUR-SEGMENT(p1, p2, p3) alors
21 |  |  Retourner Vrai ;
22 |  sinon si d4 = 0 et SUR-SEGMENT(p1, p2, p4) alors
23 |  |  Retourner Vrai ;
24 |  sinon Retourner Faux ;
25 fin
```