Licence Informatique ISTIC
Module ALG *2015-2016*

Méthodes algorithmiques
TD 1 - Diviser pour régner
(2 séances)

Concevoir des solutions basées sur la méthode étudiée

Exercice 1 (Plus grande valeur d'un tableau d'entiers non trié)
On souhaite déterminer la plus grande valeur d'un tableau de n entiers non trié TAB. Proposer un algorithme et évaluer sa compléxité.

Exercice 2 (Les tours de Hanoï)
Les tours de Hanoï sont un jeu composé de n *disques* de diamètres distincts et de trois *tiges* verticales posées sur un socle. Dans l'état initial, comme illustré dans la Figure 1 pour $n = 4$, les disques sont disposés autour de la même tige (que l'on nommera A), dans l'ordre de leur diamètre.

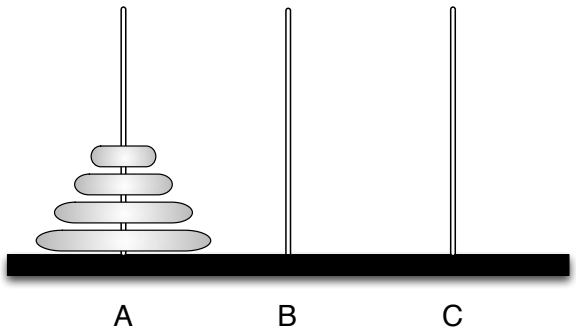


Figure 1: Les tours de Hanoï (n = 4, état initial).

Le but du jeu est de déplacer l'ensemble des disques de la tige A à la tige B en respectant les contraintes suivantes:

- La seule opération autorisée est le déplacement d'un disque d'une tige vers une autre.
- Un disque ne doit jamais reposer sur un disque de diamètre inférieur.

1. Ecrire un algorithme récursif permettant de résoudre ce problème
2. Donner sa complexité

Exercice 3 (Pavage avec des triominos)
On appelle *table à trou de dimension* $n \in \mathbb{N}$ un quadrillage $n \times n$ auquel on a retirée une case, comme illustré sur la Figure 2.

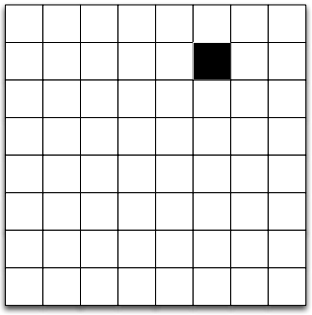


Figure 2: Une table à trou de taille 8×8

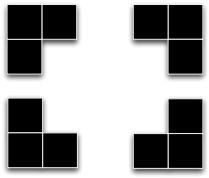


Figure 3: Les quatre triominos

On dispose de pièces, appelées *triominos* dont la forme est obtenue à partir d'une table 2×2 à laquelle on a retiré une case. Comme n'importe quelle case peut être retiré, il existe quatre triominos, illustrés par la Figure 3. On souhaite écrire un algorithme qui résoud le problème suivant:

Problème PavageTriominos

Entrée : une table à trou T
Sortie : un pavage de T avec des triomoninos

1. On note $Pav(T)$ la propriété "la table à trou T peut être pavée". Montrer par récurrence sur n que pour tout $n \in \mathbb{N}$ tel que $n = 2^r$ ($r \in \mathbb{N}$), et toute table à trou T de dimension n, on a $Pav(T)$.

2. Proposer une méthode fondée sur le principe DR permettant de résoudre le problème si n est une puissance de 2 ($n = 2^r$).

3. Dans le cas général ($\forall n \in \mathbb{N}$), quelle propriété sur n est nécessaire pour que le problème admette une solution ? En déduire, lorsque le problème admet une solution, le nombre de triominos utilisés.

Exercice 4 (Point dans un polygone)
On considère un polygone convexe (angles internes $< 180^o$) à n ($n \geq 3$) sommets, illustré par la Figure 4. Soit le pseudo-code suivant qui déclare le polygone sous la forme du tableau de points S formant son périmètre:

const $n = \ldots$;
type $Point =$ **struct int** x, **int** y **fstruct** ;
var $[1 \ldots n]$ **point** S;

On veut déterminer si un point p donné du plan est inclus (au sens large, c'est-à-dire en acceptant que le point soit sur le périmètre) dans le polygone $S[1 \dots n]$. On souhaite donc résoudre le problème suivant:

Problème PointDansPolygone

Entrée : un point p et un polygone S
Sortie : la valeur de l'affirmation "Le point p est inclus dans le polygone S."

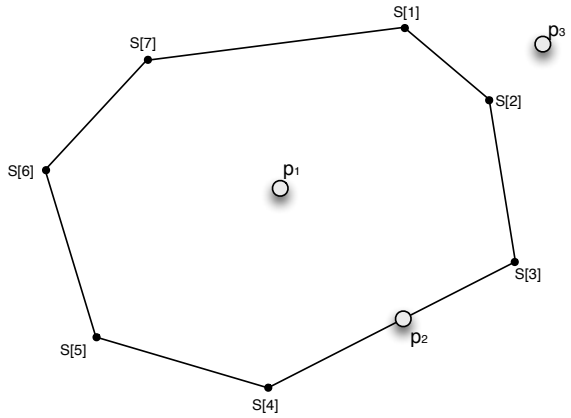


Figure 4: Un polygone convexe à 7 côtés. p_1 et p_2 y sont inclus, p_3 ne l'est pas.

On suppose donnée la fonction *même-côté* qui retourne *vrai* si les points X et Y sont situés du même côté (au sens large) par rapport à la droite (AB) avec A et B distincts.

bool *même-côté* (**Point** A, **Point** B, **Point** X, **Point** Y)

On supposera qu'un appel à la fonction *même-côté* a une complexité temporelle en $O(1)$.

1. Écrire une fonction *dans-triangle* déterminant si le point p est inclus dans le triangle formé par les points A, B, et C.

   bool *dans-triangle* (**Point** A, **Point** B, **Point** C, **Point** p)

2. Écrire une fonction

   bool *inclus1* (**int** i, **Point** p)

   qui procède par réduction simple, et dont l'appel $inclus1(n, p)$ retourne la valeur de l'assertion "le point p est inclus dans le polygone $S[1 \dots n]$".

   Quel est l'ordre de grandeur de la complexité temporelle de votre algorithme pour la fonction $inclus1$?

3. Écrire une fonction

   bool *inclus2* (**int** i, **int** j, **point** p)

   qui procède par réduction logarithmique, dont l'appel inclus2$(2, n, p)$ retourne la valeur de l'assertion "le point p est inclus dans le polygone $S[1 \dots n]$".

   Quel est l'ordre de grandeur de la complexité temporelle de algorithme pour la fonction $inclus2$?

Exercice 5 (Recherche d'un pic dans un tableau d'entiers non trié A)

Soit le tableau d'entiers non trié $A[0, ..., n-1]$. On supposera $A[-1] = A[n] = -\infty$.

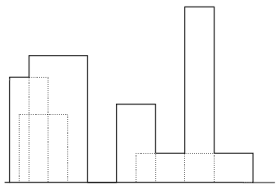
Definition 1 *La case $A[i]$ est dite pic si $A[i-1] \leq A[i] \geq A[i+1]$.*

On souhaite trouver au moins un pic du A. Proposer un algorithme et évaluer sa compléxité.

Exercice 6 (Recherche d'un pic dans un tableau carré d'entiers A)
Soit le tableau d'entiers $A[n, n]$.

Definition 2 *La case $A[i, j]$ est dite pic si $A[i-1, j] \leq A[i, j] \geq A[i+1, j]$ et $A[i, j-1] \leq A[i, j] \geq A[i, j+1]$.*

Exercice 7 (Le problème des gratte-ciels)
La description de n gratte-ciels est représentée dans un tableau I, chaque immeuble est représenté par un intervalle de points $[a, b]$ désignant sa ligne d'horizon (donc $a.y = b.y$).
Proposer un algorithme pour décrire la ligne qui décrit les toits des gratte-ciels pour un observateur qui regarde le paysage en face urbain ; certaines parties d'immeubles sont cachées.
Evaluer la complexité.



[h]

Exercice 8 (La distance de Levenshtein (distance d'édition, de similarité))

- La distance de Levenshtein entre mots ou chaînes de caractères donne des indications sur le degré de ressemblance de ces chaînes.

Exemple : Deux alignements possibles des mots "SNOWY" et "SUNNY".

S	-	N	O	W	Y		-	S	N	O	W	-	Y
S	U	N	N	-	Y		S	U	N	-	-	N	Y
coût = 3							coût = 5						

- Le symbole "_" indique "gap" (omission). On peut en utiliser tant que l'on veut.
- Le coût d'un alignement équivaut le nombre de colonnes où les caractères divergent.
- La distance de Levenshtein correspond au coût du meilleur alignement.
- Si A, B sont deux mots, la distance de Levenshtein est le nombre minimum de remplacements, ajouts et suppressions de lettres pour passer du mot A au mot B.

La distance de Levenshtein entre deux séquences de caractères $x[1, 2, \ldots, m]$ et $y[1, 2, \ldots, n]$ peut être calculée par la programmation dynamique en utilisant la récurrence:

$$E(i,j) = \min\{1 + E(i-1,j), 1 + E(i,j-1), \text{diff(i,j)} + E(i-1,j-1)\}. \quad (1)$$

où diff(i,j)=0, si $x[i] = y[j]$, 1 sinon.

Algorithm 1 Alignment(X,Y)

```
1: Initialize E(i,0) = i for each i
2: Initialize E(0,j) = j for each j
3: for i = 1 to m do
4:    for j = 1 to n do
         Use the recurrence (1) to compute E(i,j)
5:    end for
6: end for
7: return E(m,n)
```

1. Evaluer la complexité (en temps et en mémoire) pour calculer la valeur $E(m,n)$.

2. Evaluer la complexité (en temps et en mémoire) pour calculer l'alignement optimal.

3. Utiliser l'approche diviser pour résoudre pour réduire la complexité en mémoire de l'algorithme pour calculer l'alignement optimal.

   Suggestion:

   (a) On construit un graphe G_{XY} (orienté et valué) "grid-graph" avec des longueurs sur les arcs correspondantes au coûts des omissions (verticale), insertions (horizontale) et substitutions (diagonale). Soit $f(i,j)$ la longueur du chemin le plus court de $(0,0)$ à (i,j) dans le graphe G_{XY} (on notera PCC$((0,0) \rightsquigarrow (i,j))$. Alors $f(i,j) = E(i,j)$.

   (b) D'une manière symétrique, définir la récurrence $g(i,j)$ qui permet de calculer la valeur du PCC$((m,n) \rightsquigarrow (i,j))$.

   (c) On peut démontrer que, quelque soit la case (i,j), la longueur du PCC$((0,0) \rightsquigarrow (m,n))$ qui passe par cette case est $f(i,j) + g(i,j)$.

   (d) Soit k une colonne quelconque et soit $q = \underset{i}{\text{argmin}}\, f(i,k) + g(i,k)$. On peut démontrer qu'il existe un PCC$((0,0) \rightsquigarrow (m,n))$ qui passe par la case (q,k). Donner la complexité en temps et en mémoire pour calculer l'indice q.

   (e) Utiliser l'indice q pour concevoir un algorithme de type Diviser_pour_Résoudre qui permet de retrouver l'alignement optimal en utilisant une mémoire linéaire et ayant la même complexité en temps que l'algorithme original.