SDD
17

Ecrire les axiomes exprimant la complétude suffisante pour la fonction has.

has : BinarySearchTree [T] x T → booléen.

On exprime la complétude de has sur les fonctions internes de BinarySearchTree [T] c'est-à-dire les fonctions qui ont un résultat de type BinarySearchTree [T].

① Sur les constructeurs

① has (empty (), v) = faux
② has (makeRoot (l, v, r), v) = vrai
③ has (makeRoot (l, v₁, r), v₂) = has (l, v₂) si v₂ < v₁
④ has (r, v₂) si v₂ > v₁

② Sur deleteGreatest.

deleteGreatest (empty ()) = empty ()
deleteGreatest (makeRoot (l, v, empty ())) = l
deleteGreatest (makeRoot (l, v, r)) = makeRoot (l, v, deleteGreatest (r))
si r ≠ empty ()

On englobe les arbres avec les observateurs pour éviter l'égalité entres arbres :

⑤ has (deleteGreatest (empty ()), v) = has (empty (), v).
= faux.

⑥ has (deleteGreatest (makeRoot (l, v₁, empty ())), v₂) = has (l, v₂).
⑦ has (deleteGreatest (makeRoot (l, v, makeRoot (l₁, v₁, r₁))), v₂) =
has (makeRoot (l, v, deleteGreatest (makeRoot (l₁, v₁, r₁)))), v₂).

③ Sur delete :

① delete (empty(), v) = empty().

② delete (makeRoot (l, v, r), v) =
makeRoot (deleteGreatest (l), greatest (l), r).
si l ≠ empty().

③ delete (makeRoot (empty(), v, r), v) = r.

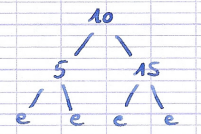
delete (makeRoot (l, v_1, r), v_2) =

④ | makeRoot (delete (l, v_2), v_1, r) si $v_1 > v_2$.

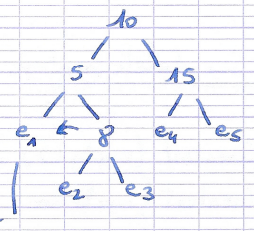
⑤ | makeRoot (l, v_1, delete (r, v_2)) si $v_1 < v_2$.

Adjonction à la racine dans un arbre binaire de recherche.

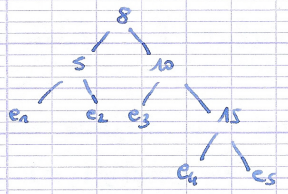
idée : ajouter aux feuilles et "le remonter" à la racine.



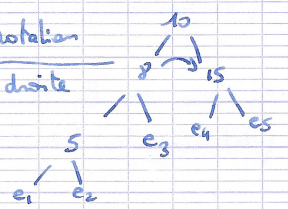
ajout de "8" :



rotation à gauche



rotation droite

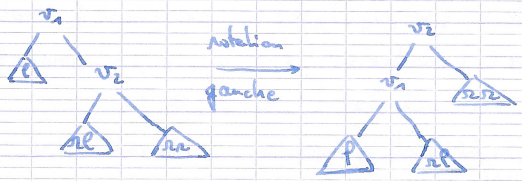


SDD

18

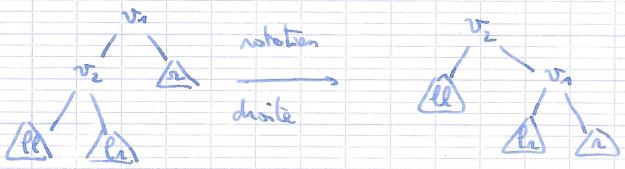
Définition des rotations gauche et droite : leftRotate, rightRotate.

Axiomes:



leftRotate (makeRoot (l, v_1, makeRoot (rl, v_2, rr))) =
makeRoot (makeRoot (l, v_1, rl), v_2, rr).

leftRotate (b) défini ssi hasRight (b) (b a un sous-arbre droite non vide).



rightRotate (makeRoot (makeRoot (ll, v_2, lr), v_1, r)) =
makeRoot (ll, v_2, makeRoot (lr, v_1, r)).

rightRotate (b) défini ssi hasLeft (b) (b a un sous-arbre gauche non-vide)

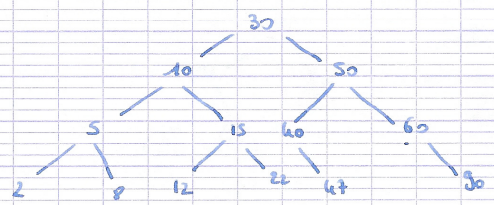
Remarque : leftRotate et rightRotate préserve la structure d'arbre binaire de recherche.

Ecrire les axiomes de insertRoot.

insertRoot (empty(), v) = makeRoot (empty(), v, empty()).
insertRoot (makeRoot (l, v, r), v) = makeRoot (l, v, r).
insertRoot (makeRoot (l, v_1, r), v_2) =
    rightRotate (makeRoot (insertRoot (l, v_2), v_1, r)). si $v_2 < v_1$.
    leftRotate (makeRoot (l, v_1, insertRoot (r, v_2))) si $v_2 > v_1$.

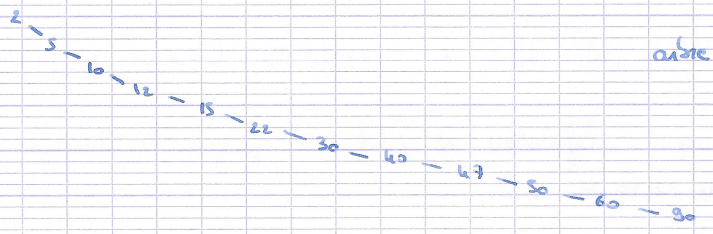
Exercices:

a) Construire un ABR par adjonctions aux feuilles : 30, 10, 50, 15, 40, 60, 5, 2, 47, 22, 12, 8, 90.



arbre bien équilibré.

b) Même question avec : 2, 5, 8, 10, 12, 15, 22, 30, 40, 47, 50, 60, 90.



arbre "dégénéré" en une (double) liste.

pour chercher le nombre 100 : ① 4 comparaisons : dans le pire des cas : $O(\log_2(n))$
② 11 comparaisons : dans le pire des cas : $O(n)$
$O($ hauteur de l'arbre $)$

SDD
10

III] Arbres AVL : (Adelson Velskii et Landis). en 1960.

Définition : **Arbre H-équilibré.**

Un arbre binaire est H-équilibré si en tout noeud de l'arbre les hauteurs des sous-arbres gauche et droit diffèrent d'au plus 1.

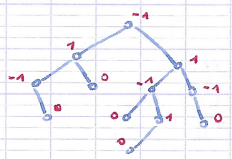
Spécification :

desequilibre (empty()) = 0.
desequilibre (makeRoot (l, v, r)) = height (l) − height (r).

Un arbre t est H-équilibré ssi pour tout sous-arbre s de t
desequilibre (s) $\in \{-1, 0, 1\}$

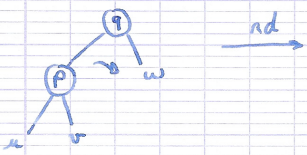
Définition : Un A.V.L. est un arbre binaire de recherche H-équilibré.

Exemple d'arbre équilibré.

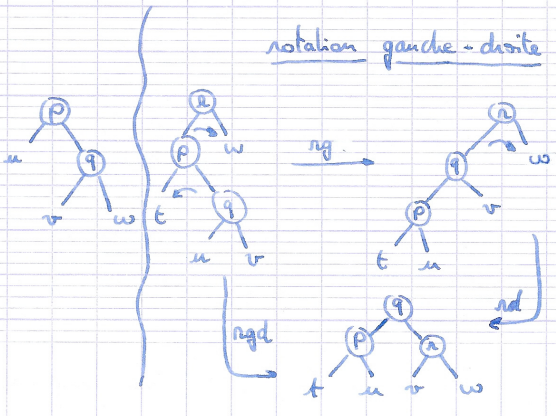


Définitions des rotations :

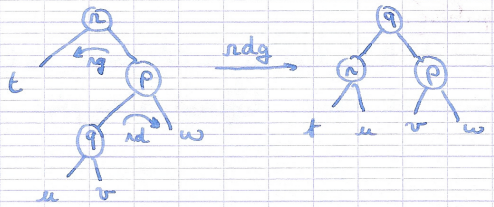
rotation droite :



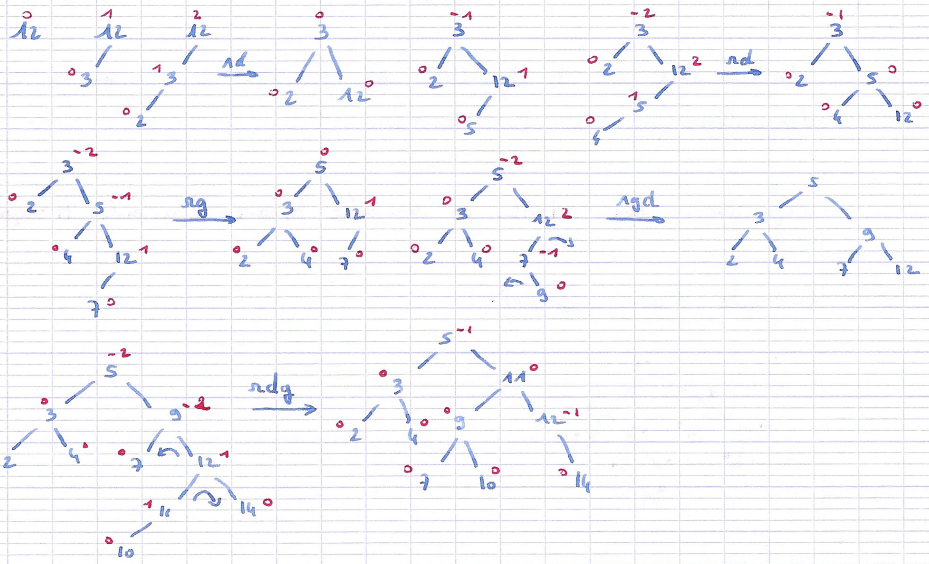
rotation gauche - droite



Rotation droite - gauche :



Adjonction dans un A.V.L : 12, 3, 2, 5, 4, 7, 9, 11, 14, 10.



SDD
2o

Principe général de l'équilibrage des AVL.

Soit t = makeRoot (g, r, d). On suppose qu'on ajoute x sur une feuille de g.
Cette adjonction fait augmenter la hauteur de g de 1.
g reste un AVL (si ce n'était pas le cas, on travaillerait sur des sous-arbres)

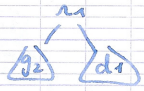
Montrons que : avant l'adjonction de x le déséquilibre de g vaut 0.
Démonstration :

hypothèses :
- g AVL avant adjonction
- g reste un AVL après
- la hauteur de g a augmenté

Conclusion : le déséquilibre de g est 0 avant l'adjonction de x.

g AVL avant : supposons desequilibre$(g) = 1$

avant : $g =$ 

Comme hauteur ↑ de 1 après $g =$ 

$g_2 = g_1 \cup \{x\}$ déséq = 2

impossible car g doit rester un AVL
si desequilibre$(g) = -1$ $\Rightarrow$ ↑ raisonnement impossible
donc desequilibre$(g) = 0$.

Distinguons les cas suivants :

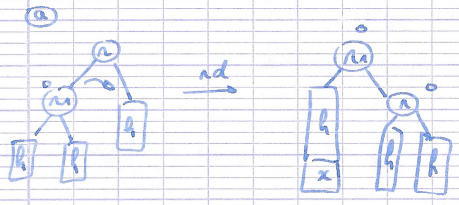
① Si desequilibre(t) = 0 avant
desequilibre(t) = 1 après.
} t reste un AVL et sa hauteur augmente de 1.

② Si desequilibre(t) = −1 avant
desequilibre(t) = 0 après
} et sa hauteur reste inchangée

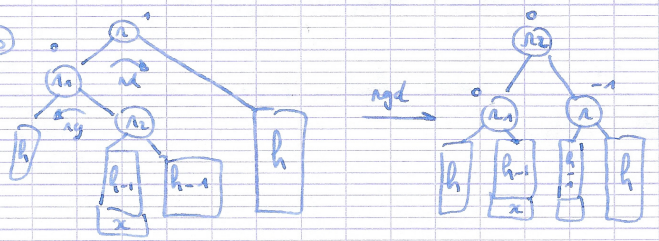
③ Si déséquilibre(t) = 1 avant
desequilibre(t) = 2 après
} t n'est plus un AVL, il faut le rééquilibrer.

Trois cas se présentent :

ⓐ



ⓑ



SDD

21

Axiomes concernant l'adjonction dans les AVLs.

opérations :

ajouterAVL : AVL[T] × T → AVL[T]

reequilibrer : AVL[T] → AVL[T]

desequilibre : AVL[T] → entier.

précondition :

reequilibre(t) est défini ssi desequilibre(t) ∈ {-2, -1, 0, 1, 2}

Axiomes sur ajouterAVL :

ajouterAVL(empty(), v) = makeRoot(empty(), v, empty()).

ajouterAVL(makeRoot(l, v1, r), v2) =

| reequilibrer(makeRoot(ajouterAVL(l, v2), v1, r)) si v2 < v1.

| reequilibrer(makeRoot(l, v1, ajouterAVL(r, v2))) si v2 > v1.

ajouterAVL(makeRoot(l, v, r), r) = makeRoot(l, v, r).

Axiomes sur reequilibrer

précondition :

reequilibrer(t) = t si desequilibre(t) ∈ {0, -1, 1}.

reequilibrer(t) = rd(t) si desequilibre(t) == 2 et desequilibre(left(t)) == 1

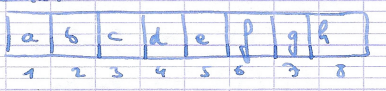
reequilibrer(t) = rgd(t) si desequilibre(t) == 2 et desequilibre(left(t)) == -1

reequilibrer(t) = rg(t) si desequilibre(t) == -2 et desequilibre(right(t)) == -1

reequilibre(t) = rdg(t) si desequilibre(t) == -2 et desequilibre(right(t)) == 1.

Implantation des arbres binaires.

- par des tableaux :



noeud étiquette "l'indice n
ses 2 fils respectifs venant à l'indice 2n et 2n+1.

- chaînée (pointeurs).

BinarySearchTree < T >
value de Type T
left, right) de type BinarySearchTree < T >.

- Représentation propre au langage à objet (avec l'héritage).

Interface de Arbre (classe abstraite).
Feuille : pas de fils
Fg : seulement 1 fils gauche
Fd : ——— droit
Fgd : 1 fils gauche et 1 fils droit.

SDD
22

Chapitre Graphes.

I] Définitions et exemples.

Graphe orienté : $G = (S, A)$ est un graphe orienté. S un ensemble fini de sommets. A un ensemble fini de couples de sommets appelés arcs.

Graphe non orienté : A : arêtes : ensemble de paires de sommets

Graphe valué (orienté ou non) : $G = (S, A, C)$ avec C une application de A vers $\mathbb{R}$.

Remarque : Dans ce cours, on considère
- les graphes sans boucles (arcs ou arêtes reliant un sommet à lui-même)
- les graphes simples ≡ au plus un arc ou une arête entre deux sommets (≠ multi-graphes).

II] Terminologie

x, y deux sommets. $x \to y$ est l'arc (x, y)
x extrémité initiale, y extrémité finale.
y successeur de x
idem pour non orienté : $x - y$ l'arête $\{x, y\}$

Dans les graphes orientés :
- demi-degré extérieur (resp intérieur) d'un sommet est égal au nombre d'arcs partant (resp aboutissant) à ce sommet.

III] Spécification algébrique. (Graphe orienté).

Graphe [T], les noeuds sont étiquetés par des éléments de type T.

Opérations :

constructeurs
- grapheVide : → Graphe [T].
- ajouterSommet : Graphe [T] × T → Graphe [T]
- ajouterArc : Graphe [T] × T × T → Graphe [T].

observateurs
- estVide : Graphe [T] → booléen
- estUnSommet : Graphe [T] × T → booléen
- estUnArc : Graphe [T] × T × T → booléen.

Convention : - Ajout d'un sommet : c'est un sommet isolé
- Ajout d'un arc ⇒ les sommets adjacents sont dans le graphe
- Retrait d'un sommet ⇒ tous les sommets adjacents sont supprimés.

observateurs
- demiDegreExterieur : Graphe [T] × T → entier
- demiDegreInterieur : Graphe [T] × T → entier.
- miemeSuccesseur : Graphe [T] × T × entier → T
- miemePredecesseur : Graphe [T] × T × entier → T

Fonctions internes (ici destructeurs)
- retirerSommet : Graphe [T] × T → Graphe [T]
- retirerArc : Graphe [T] × T × T → Graphe [T].

Préconditions :

ajouterSommet (g, s) est défini ssi non (estUnSommet (g, s)).
ajouterArc (g, s1, s2) est défini ssi non (estUnArc (g, s1, s2)).
demiDegreExterieur (g, s) est défini ssi estUnSommet (g, s).
demiDegreInterieur (g, s) est défini ssi estUnSommet (g, s).

mièmeSuccesseur (g, s, m) est défini ssi
est UnSommet (g, s) et m ≤ demiDegreExterieur (g, s).

mièmePredecesseur (g, s, m) est défini ssi
estUnSommet (g, s) et m ≤ demiDegre_Interieur (g, s).

retirerSommet (g, s) défini ssi estUnSommet (g, s).
retirerArc (g, s1, s2) défini ssi estUnArc (g, s1, s2).

Axiomes :

Sur estVide : à écrire.
Sur estUnSommet :

estUnSommet (grapheVide (), s) = faux.
estUnSommet (ajouterSommet (g, s), s) = vrai
estUnSommet (ajouterSommet (g, s1), s2) = estUnSommet (g, s2) si s1 ≠ s2.
estUnSommet (ajouterArc (g, s1, s2), s3) = estUnSommet (g, s3)

Sur estUnArc :

estUnArc (grapheVide (), s1, s2) = faux
estUnArc (ajouterSommet (g, s1), s2, s3) = estUnArc (g, s2, s3).
estUnArc (ajouterArc (g, s1, s2), s1, s2) = vrai
estUnArc (ajouterArc (g, s1, s2), s3, s4) = estUnArc (g, s3, s4) si s1 ≠ s3
et s2 ≠ s4.

Ecrire les axiomes pour la fonction demiDegreExterieur.

demiDegreExterieur (ajouterSommet (g, s), s) = 0

demiDegreExterieur (ajouterSommet (g, s1), s2) = demiDegreExterieur (g, s2)
si s1 ≠ s2.

demiDegreExterieur (ajouterArc (g, s1, s2), s3) = demiDegreExterieur (g, s3)
si s1 ≠ s3.
et s2 ≠ s3

demiDegreExterieur (ajouterArc (g, s1, s2), s1) =
demiDegreExterieur (g, s1) + 1

IV] Représentations des graphes.

a) Matrice d'adjacence (booléen).
b) Liste d'adjacence : 1 sommet → liste de ses successeurs

V] Parcours de graphes.

a) En profondeur : On part d'un sommet, on va au plus profond en parcourant les successeurs des sommets

parcours : Graphe [T] → Liste [T]

l : liste comptant les sommets du graphes.

```
parcoursProfondeur (G : graphe, s : sommet)
  l = addLast (l, s); marque [s] = Vrai ;
  pour j de 1 à demiDegreExterieur (G, s)
faire | r = iemeSuccesseur (g, s, j)
      | si non (marque [r])
      | alors parcoursProfondeur (G, r)
ffaire
```

SDD
24

Programme Principal

```
pour i de 1 à n faire
        marque[i] = Faux
ffaire
fpour

P = emptylist
pour i de 1 à n faire
    si non (marque[i])
    alors parcoursProfondeur (G, i)
    fsi
ffaire
fpour
```

SDD
25

Graphes : Plus courts chemins

I] Introduction.

Applications : Réseaux, ordonnancement
Contexte : Graphes orientés valués.
$G = (S, A, Cost)$

Conditions d'existence d'un plus court chemin de s_1 à s_2.
- Existence d'un chemin de s_1 à s_2.
- Pas de circuit absorbant (ayant un coût négatif).

[cf : Théorie des Graphes (HSED, IAI)]

Algorithme de Dijkstra

On suppose que les coûts des arcs sont positifs ou nuls.

Données : $G = (S, A, coût)$, $s \in S$ sommet donné.

- CC : ensemble des sommets pour lesquels on connaît un plus court chemin
- M : le complémentaire de CC dans S.
- d : tab [1..n]
  d[i] coût d'un plus court chemin de i à CC.
- pr : tab [1... n]
  pr[i] est le prédecesseur du sommet i dans un plus court chemin dans CC de s à i.

Algo: INIT.
jusqu'à ARRET, faire CORPS

INIT : $CC := \{s\}$
$M := S - \{s\}$
$\forall i \in [1,n]$, $d[i] :=$ si l'arc (s,i) existe alors $cout((s,i))$ sinon $+\infty$.

$\forall i \in [1,n]$, $pr[i] := s$

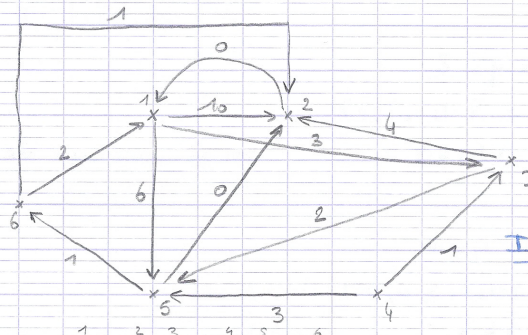
CORPS :
- On retire de M un élément y tel que $d[y]$ soit minimal
- On ajoute y à CC.
- On met à jour $d[u]$ et $pr[u]$ pour les successeurs immédiats u de y pour lesquels $d[y] + cout((y,u)) < d[u]$

ARRET : $M = \emptyset$ ou $(\forall x \in M)\ d[x] = +\infty$

SDD
26

Exemple : G , S = {1, 2, 3, 4, 5, 6}

Coût = {((1,2), 10), ((1,3), 3), ((1,5), 6), ((2,1), 0),
((3,2), 4), ((3,5), 2), ((4,3), 1), ((4,5), 3),
((5,2), 0), ((5,6), 1), ((6,1), 2), ((6,2), 1)}



Execution : (on cherche les plus courts chemins partant du sommet 1)

INIT : CC = {1}
M = {2, 3, 4, 5, 6}

	1	2	3	4	5	6
d	0	10	3	∞	6	∞
pr	1	1	1	1	1	1

Etape 1 : Choix du sommet 3.
CC = {1, 3}
M = {2, 4, 5, 6}

d :	0	7	3	∞	5	∞
pr :	1	3	1	1	3	1

d[3] + cout((3,5)) < d[5]
3 + 2 < 6

d[y] + cout((y,u)) < d[u]
4 + 3 < 10

Etape 2 : choix $y = 5$.

$CC = \{1, 3, 5\}$

$M = \{2, 4, 6\}$

successeurs immédiat de 5 dans M : 2 et 6

0	5	3	∞	5	6

1	5	1	1	3	5

Etape 3 : choix $y = 2$

$CC = \{1, 2, 3, 5\}$

$M = \{4, 6\}$

pas de successeurs de 2 dans M.

Etape 4 : choix $y = 6$

$CC = \{1, 2, 3, 5, 6\}$

$M = \{4\}$.

d

0	5	3	∞	5	6

pas de successeurs de 6 dans M.

pr

1	5	1	1	3	5

Ecrire les plus courts chemin de 1 vers x

$x = 1$: 1

$x = 2$: $1 \to 3 \to 5 \to 2$: valeur $3 + 2 + 0 = 5$

$x = 3$: $1 \to 3$: valeur : 3.

$x = 4$: pas de plus court chemin de 1 vers 4.

$x = 5$: $1 \to 3 \to 5$: valeur $3 + 2 = 5$

$x = 6$: $1 \to 3 \to 5 \to 6$: valeur : $3 + 2 + 1 = 6$.

SDD
27

Analyse de la complexité de l'algorithme.

INITIALISATION : n sommets et p arcs

```
CC := {s}
M := S - {s}
pour i de 1 à n, faire
    d[i] := cout((s,i)).
    pr[i] := s.
```

$O(n)$

```
jusqu'à ARRET, faire
    y := distance_min(M, d)
    M := M - {y}
    CC := CC ∪ {y}
    pour u ∈ [successeur(y) - CC], faire
        si d[y] + Cout((y,u)) < d[u].
        alors | d[u] := d[y] + cout((y,u))
              | pr[u] := y
        fsi
    fpour
fjusqu'à
```

→ la boucle effectuée au plus $(n-1)$ fois.

→ calcul du minimum nécessite $|M| - 1$ comparaisons d'où $O(n^2)$ comparaisons

→ globalement le nombre d'opérations est majoré par p $O(p)$

$p \leq n^2$ donc $O(n^2)$