ASI3

Algorithmique
et
Base de la programmation

Correction

1 Tri rapide (3 points)

1.1 Principe

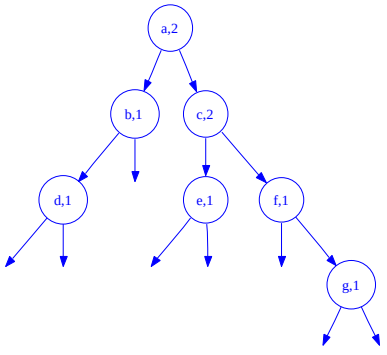
Voir le cours

2 Le nombre de Strahler

Le nombre de Strahler d'un arbre binaire est défini par :

$$\Phi(A) = \begin{cases} 0 & \text{si l'arbre } A \text{ est vide} \\ max(\Phi(A_g), \Phi(A_d)) & \text{si } A \text{ est non vide et } \Phi(A_g) \neq \Phi(A_d) \\ 1 + \Phi(A_d) & \text{si } A \text{ est non vide et } \Phi(A_g) = \Phi(A_d) \end{cases}$$

1. Donnez le TAD ArbreBinaire vu en cours
   Voir le cours



2.

3. Donnez un algorithme permettant de calculer $\Phi(A)$

```
fonction nbDeStralher (a : ArbreBinaire) : Naturel
  Déclaration  nbAg, nbAd : Naturel
debut
  si estVide(a) alors
    retourner 0
  sinon
```

```
    nbAg ← nbDeStralher(obtenirFilsGauche(a))
    nbAd ← nbDeStralher(obtenirFilsDroit(a))
    si nbAg≠nbAd alors
      retourner max(nbAg,nbAd)
    sinon
      retourner 1+nbAd
    finsi
  finsi
fin
```

3 Un peu de C

Identifiez les 6 erreurs se trouvant dans le programme ci-dessous. Indiquer brièvement la façon de les corriger.

```
1  #include"stdio.h"
2
3  int main() {
4    char * chaine, motDePasse="Mot2Passe";
5    scanf("%s",chaine);
6    if (chaine == motDePasse)
7      printf("Vous pouvez passer.\n")
8      return 0;
9    else
10     printf("C'est pas ca\n");
11     return 1;
12 }
```

- ligne 1 : < et > à la place de ""
- ligne 5 : `chaine` non allouée
- ligne 4 : manque `char *` pour `motDePasse`
- ligne 6-8 : manque les { et }
- ligne 7 : manque ;
- ligne 6 : comparaison qui ne marchera jamais (il faut utiliser `strcmp`)

4 Comment retrouver son chemin ? (10 points)

L'objectif de cet exercice est d'étudier le problème du labyrinthe. Comme l'indique la figure 1, l'objectif est de trouver un algorithme permettant de trouver le chemin qui mène de l'entrée à la sortie.

4.1 Partie publique

En fait, un labyrinthe est composé de cases. On accède à une case à partir d'une case et d'une direction. Les directions possibles sont `Nord`, `Sud`, `Est` et `Ouest`.

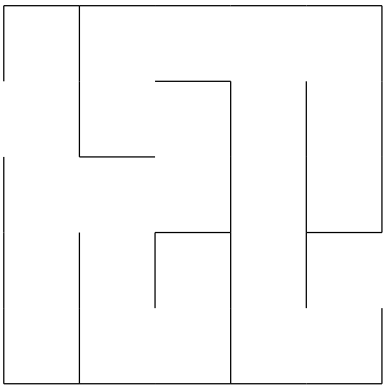


FIG. 1 – Un labytinthe

Par exemple, comme le montre la figure 2 le labyrinthe précédent peut être considéré comme étant composé de 25 cases. La case numéro 6 est la case d'entrée. La case 20 est la case de sortie. La case 8 est accessible depuis la case 13 avec la direction `Nord`.

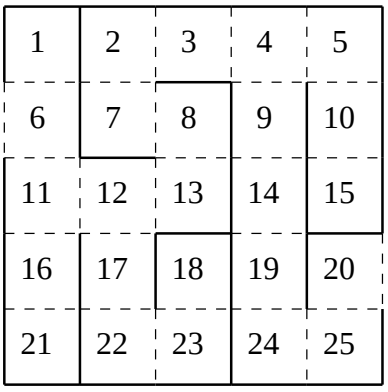


FIG. 2 – Un labytinthe composé de cases

4.1.1 Le TAD labyrinthe

Les opérations disponibles sur un labyrinthe sont les suivantes :
- créer un labyrinthe,
- obtenir la case d'entrée,
- savoir si une case est la case de sortie,
- obtenir une liste de directions possibles depuis une case donnée,
- obtenir la case accessible depuis une case avec une direction.

1. Donnez le type `Direction`
   Type Direction = {Nord,Sud,Est,Ouest}
2. Donnez le TAD `Labyrinthe` TAD liés donc définis en même temps comme nous l'avons vu en TD)
   Nom: Labyrinthe,Case
   Utilise: Direction

Opérations:	labyrinthe:	**Naturel** × **Naturel** → Labyrinthe
	caseDEntree:	Labyrinthe → Case
	estCaseDeSortie:	Labyrinthe × Case → **Booleen**
	directionsPossibles:	Labyrinthe × Case → Liste<Direction>
	caseDestination:	Labyrinthe × Case × Direction → Case

4.1.2 Algorithme du petit-poucet

Une solution pour trouver la sortie est d'utiliser le principe du petit poucet, c'est-à-dire mettre un caillou sur les cases rencontrées.

Pour ne pas modifier le TAD Labyrinthe, plutôt que de marquer une case avec un caillou on peut ajouter une case à un ensemble. Pour vérifier si on a déja rencontré une case, il suffit alors de vérifier si la case est présente dans l'ensemble.

Après avoir rappelé les TAD Ensemble et ListeChaînée vu en cours, proposer le corps de la procédure suivante qui permet de trouver le chemin de sortie (s'il existe) à partir d'une case donnée[1] :

procédure calculerCheminDeSortie (**E** l : Labyrinthe, caseCourante : Case , **E/S** casesVisitees : Ensemble<Case> , **S** permetDAllerJusquALaSortie : **Booleen**, lesDirectionsASuivre : ListeChaînée<Direction>)

procédure calculerCheminDeSortie (**E** l : Labyrinthe, caseCourante : Case , **E/S** casesVisitees : Ensemble<Case> , **S** permetDAllerJusquALaSortie : **Booleen**, lesDirectionsASuivre : ListeChaînée<Direction>)

```
Déclaration  directions : Liste<Direction>
             i : Naturel
             solutionTrouvee : Booleen
             caseTest : Case
debut
  si estCaseDeSortie(caseCourante) alors
    permetDAllerJusquALaSortie ← VRAI
    lesDirectionsASuivre ← listeChainee()
  sinon
    si non estPresent(casesVisitees,caseCourante) alors
      casesVisitees ← ajouter(casesVisitees,caseCourante)
      directions ← directionsPossibles(l,caseCourante)
      permetDAllerJusquALaSortie ← FAUX
      i ← 1
      tant que i≤longueur(directions) et non permetDAllerJusquALaSortie faire
        caseTest ← caseDestination(l,obtenirElement(directions,i))
        calculerCheminDeSortie(l,caseTest,casesVisitees,permetDAllerJusquALaSortie,
        lesDirectionsASuivre)
        si permetDAllerJusquALaSortie alors
          lesDirectionsASuivre ← ajouter(lesDirectionsASuivre,obtenirElement(directions,i))
```

[1] Vous pouvez vous inspirer de la procédure *remplir* vu dans le dernier cours

```
            finsi
            i ← i+1
        fintantque
    finsi
  finsi
fin
```

4.2 Partie privée

4.2.1 Le graphe

On peut représenter un labyrinthe à l'aide d'un graphe étiqueté et valué. On considère dans ce cas que les valeurs des nœuds du graphe sont les cases du labyrinthe et les arcs étiquetés par les directions.

Dessinez le graphe associé à l'exemple de la figure 3.

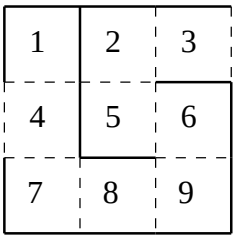
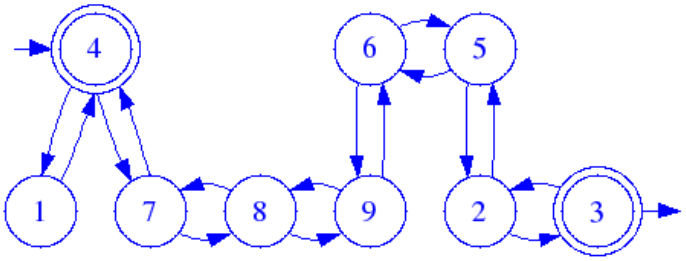


FIG. 3 – Un labytinthe composé de 9 cases



4.2.2 Représentation du graphe

Proposez la matrice d'adjascence du graphe précédent.

	1	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*
1				V					
2			V		V				
3		V							
4	V						V		
5		V				V			
6					V				V
7				V				V	
8							V		V
9						V		V	