Cours6
La récursivité

Année 2016-2017 J-F. Kamp - IUT de Vannes

PLAN

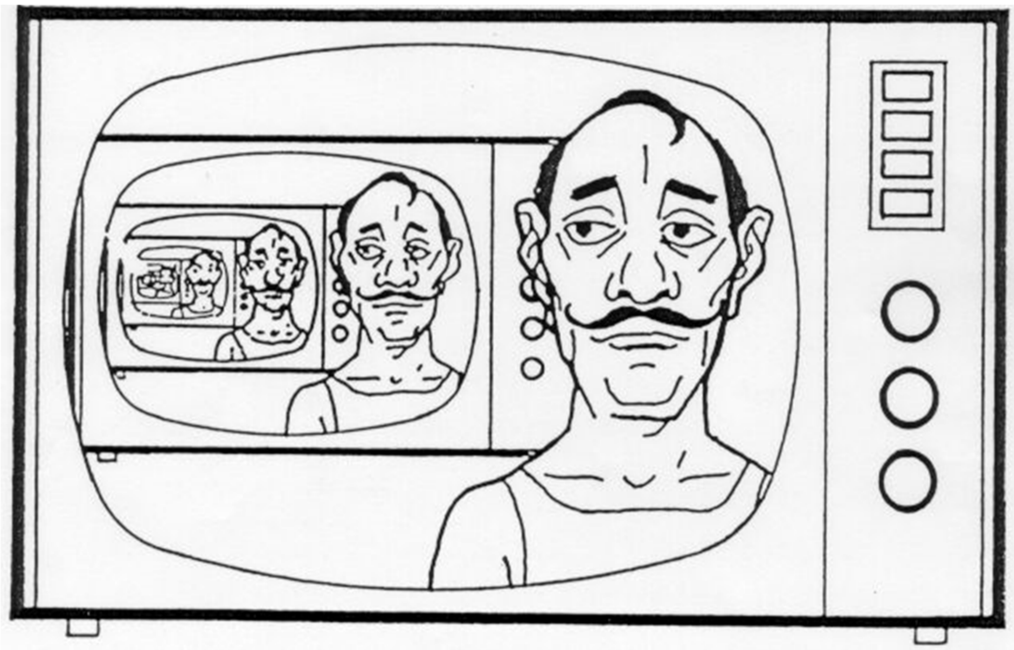
- Notion de récursivité
  - Définition
  - Un exemple simple
  - Mémoire et pile d’appel
  - Intérêt de la récursivité

- Algorithme de « backtracking » :
  - Le parcours du cavalier
  - Evaluation de la complexité
  - Organisation du code

Notion de récursivité

Définition

Un objet est récursif s'il s'utilise lui-même dans sa composition ou sa définition.



Définition

Récursivité en mathématique

- Somme des N premiers entiers positifs :
  - $\sum N = \sum (N - 1) + N$
  - $\sum 0 = 0$
- Puissance N (>=0) d'un nombre X :
  - $X^N = X^{(N-1)} * X$
  - $X^0 = 1$

On peut écrire

- Somme : F(N) = F(N-1) + N

  et F(0) = 0
- Puissance :

  F(X, N) = F(X, (N-1)) * X

  et F(X, 0) = 1

Définition

Un algorithme récursif P se compose d'un ensemble d'instructions S (ne contenant pas P) et de P **lui-même**.

En Java :

Une méthode est récursive si elle est composée d'instructions dont au moins 1 d'entre elles est la méthode **elle-même**.

Un exemple simple

Somme des N premiers entiers positifs sous forme récursive :

somEntiers(N) = somEntiers(N-1) + N

```
int somEntiers ( int n ) {

    // variable locale
    int somLoc ;

    somLoc = somEntiers( n-1 ) + n ;

    return somLoc;

}
```

Le fait que la méthode s’appelle elle-même implique **forcément** la mise en place d’une boucle qui **ne s’arrêtera PAS** SAUF si on écrit une condition d’arrêt.

Un exemple simple

Condition d’arrêt : si n égale zéro alors somEntiers(0) = 0 => le sous-programme ne doit plus s’appeler lui-même et la récursivité **doit s’arrêter**.

```
int somEntiers ( int n ) {

    // variable locale
    int somLoc ;

    if ( n = = 0 ) {

        somLoc = 0;  <= Arrêt de la récursivité !
    }

    else {

        somLoc = somEntiers ( n-1 ) + n ;

    }

    return somLoc ;

}
```

Un exemple simple

Somme des N premiers entiers positifs sous forme **itérative** :

```
int somEntiers ( int n ) {

    // variables locales
    int somLoc, i ;

    i = 1 ;
    somLoc = 0;

    // boucle AVEC condition d'arrêt
    while ( i < = n ) {

        somLoc = somLoc + i ;
        i++ ;

    }

    return somLoc ;

}
```

Mémoire et pile d'appel

Comment cela se passe-t-il en mémoire pour la méthode « somEntiers » ?

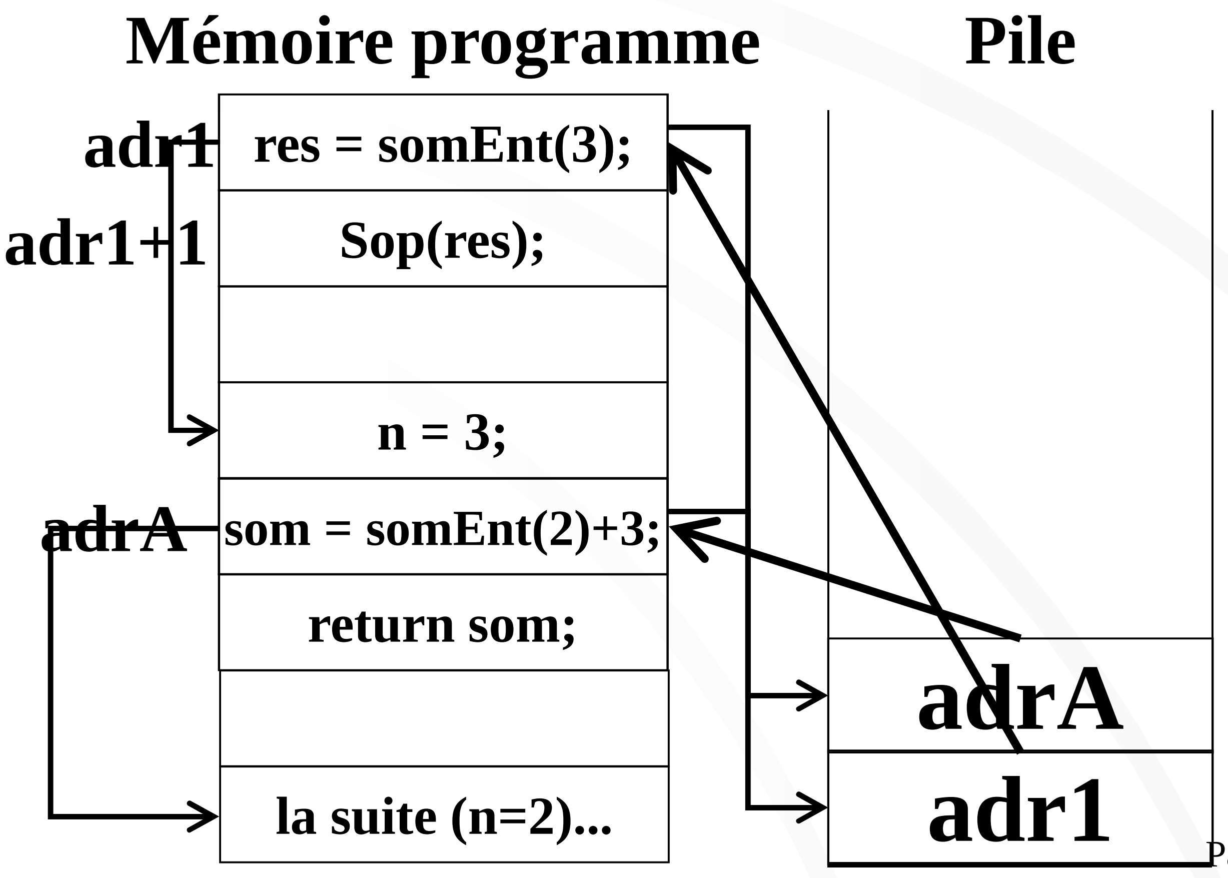
Chaque fois que le processeur rencontre le mot-clé « return » il doit savoir :

- Dans quelle variable il recopie le contenu de « somLoc » ?
- Quelle instruction de quel sous-programme il doit exécuter après ?

Ces informations sont stockées dans une pile.

Mémoire et pile d’appel

```
void principal () {
    …
    res = somEnt ( 3 ) ;
    System.out.println ( res ) ;
}

int somEnt ( int n ) {

    // variable locale
    int som;

    som = somEnt( n-1 ) + n ;
    return som;

}
```



Mémoire et pile d'appel

Mémoire programme

adr1	res = somEnt(3);
adr1+1	Sop(res);
	n = 3;
adrA	som = somEnt(2)+3;
adrA+1	return som;
	n = 2;
adrB	som = somEnt(1)+2;
adrB+1	return som;
	n = 1;
adrC	som = somEnt(0)+1;
adrC+1	return som;
adrD	n = 0;

Pile

~~adrC~~
~~adrB~~
~~adrA~~
~~adr1~~

Intérêt de la récursivité

Il est toujours possible de transformer une itération en une solution récursive et réciproquement, mais :

- Ce n’est pas toujours évident.
- Chaque solution a ses avantages et ses inconvénients.

Inconvénients de la récursivité ?

- La mémoire consommée est beaucoup plus importante que la version itérative.
- Le temps d’exécution peut être long.
- Estimation difficile de la profondeur maximale de la récursivité.

Algorithme de backtracking

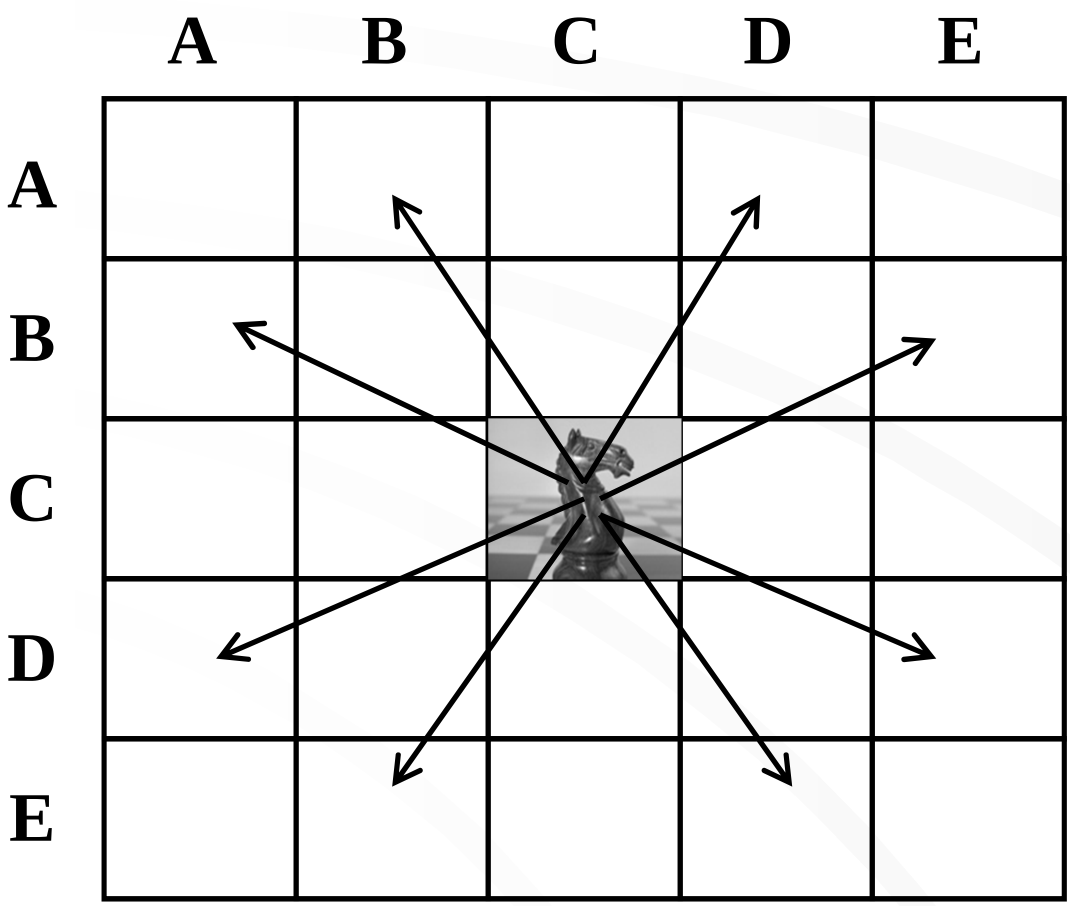
Le parcours du cavalier

Problème complexe : solution récursive (presque) obligatoire

	A	B	C	D	E
A					
B					
C					
D					
E					

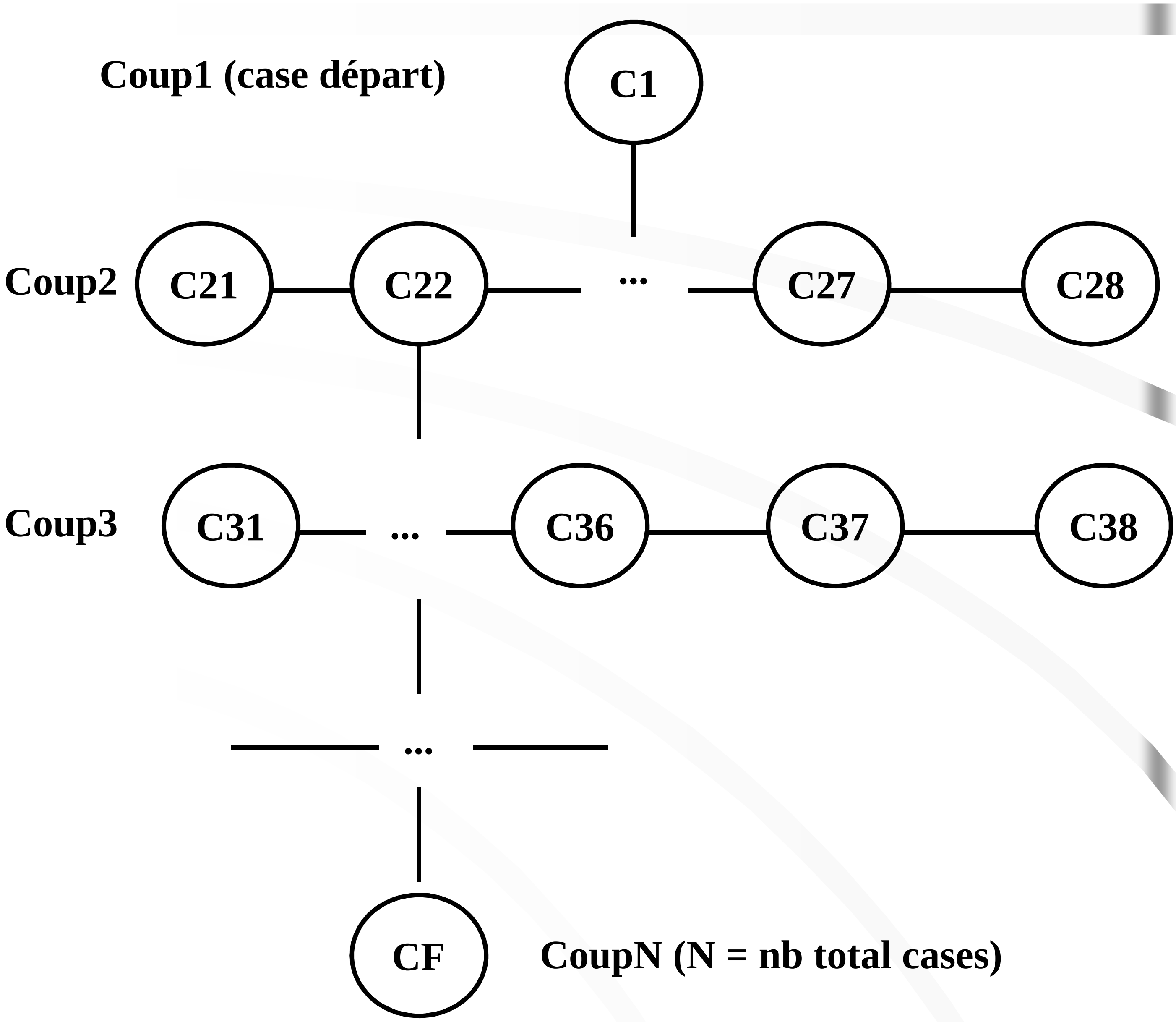
Evaluation de la complexité

Le cavalier étant sur une case valide, quels sont les déplacements possibles ?



Evaluation de la complexité

Arbre des possibilités : 8 déplacements à chaque case



Evaluation de la complexité

Pire des cas : passer par tous les chemins possibles pour ne trouver qu'en dernier lieu le seul et unique chemin qui mène à la solution (si elle existe).

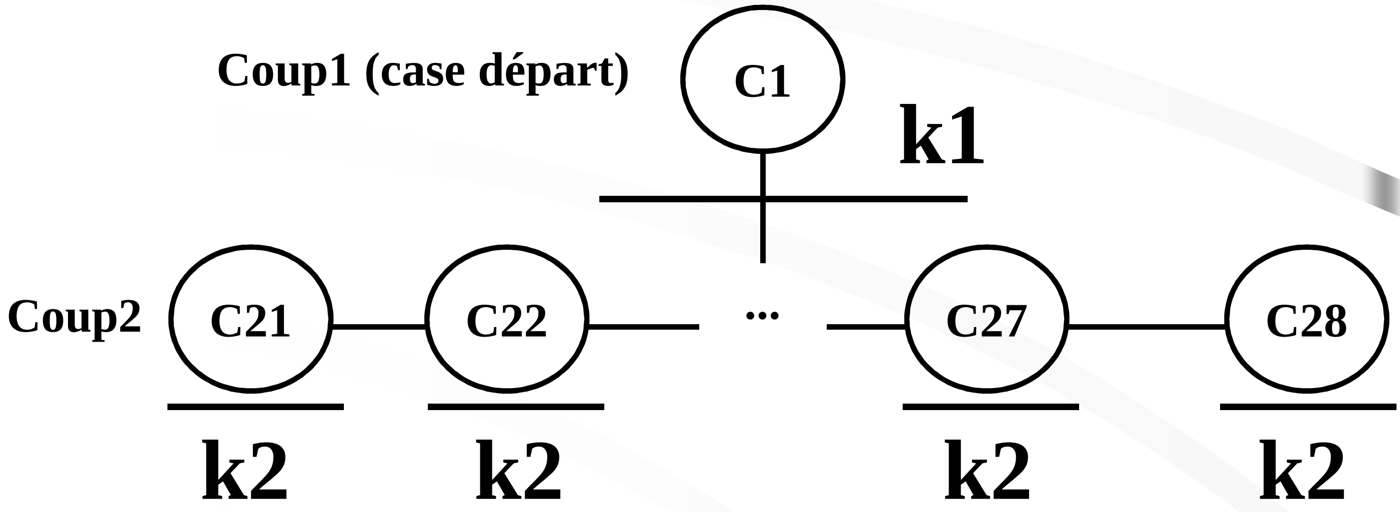
Combien de chemins à examiner ?

De quoi cela dépend-il ?

Evaluation de la complexité

A partir de la case1 (C1) : k1 chemins possibles AU TOTAL.

Soit une case du coup2 (C2i) : k2 chemins possibles.



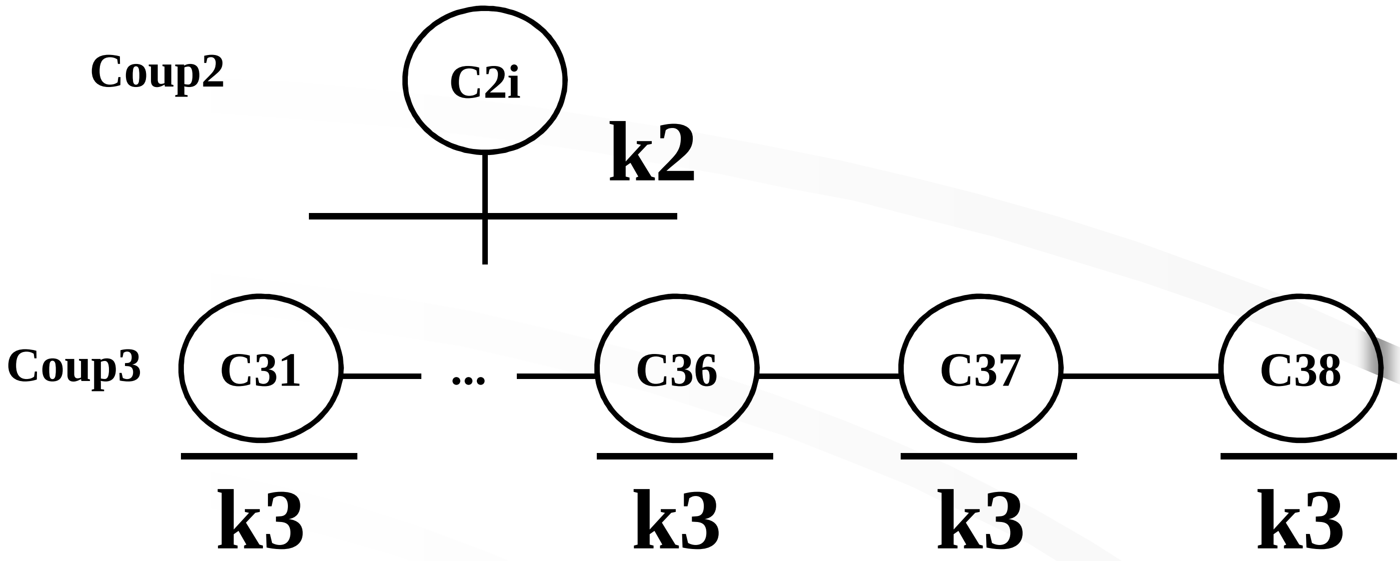
Relations entre k1 et k2 :

$$k2 < k1$$

$$k1 = 8 \text{ X } k2$$

Evaluation de la complexité

Que vaut k2 **?**



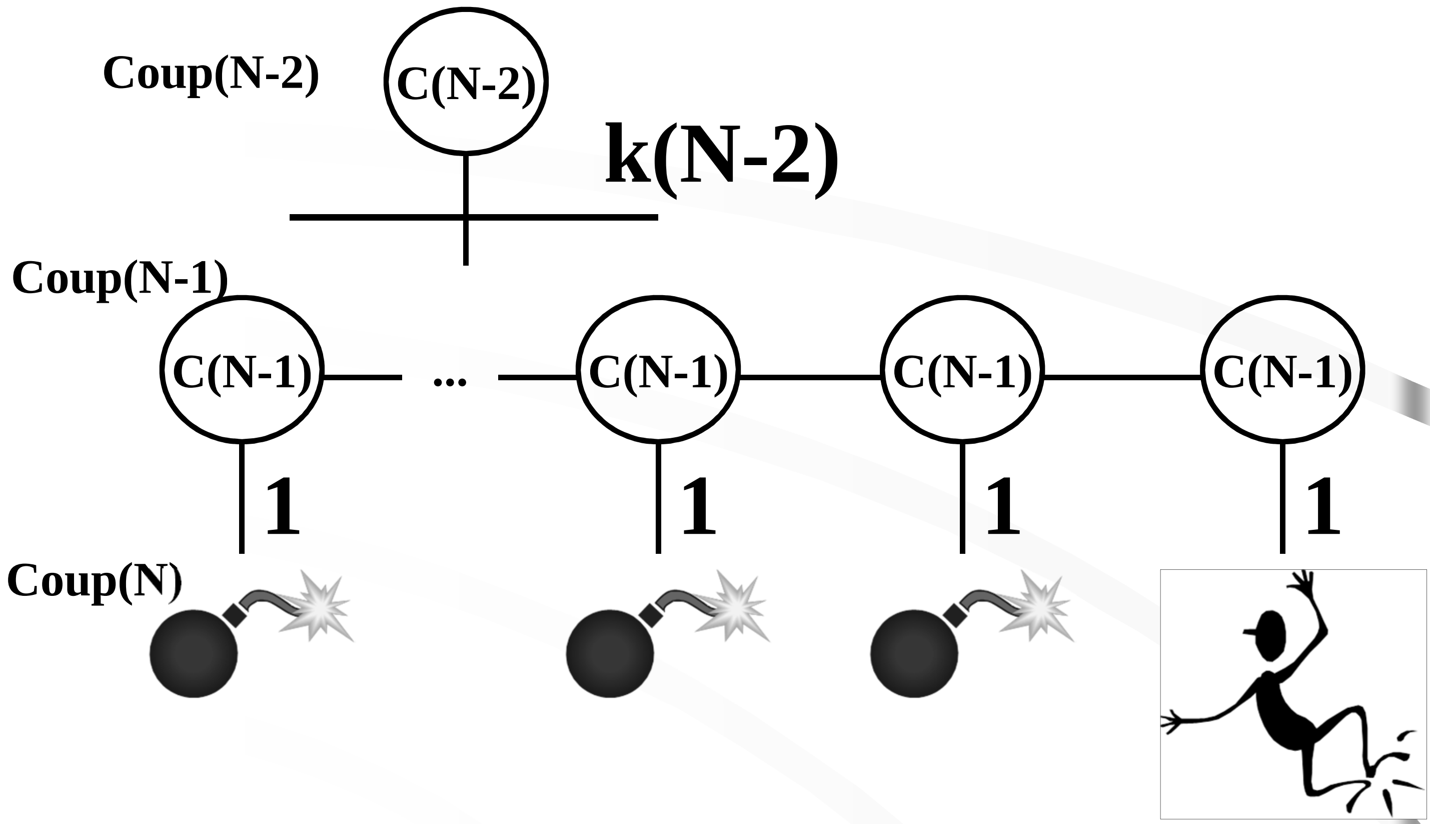
Relations entre k2 et k3 :

k3 < k2

k2 = 8 X k3

Evaluation de la complexité

Que se passe-t-il à l'avant-dernier coup ?



k(N-2) = 8 X 1 = 8

N = nbre total de cases

Evaluation de la complexité

Calcul du nombre total de chemins à explorer (au pire)

$$
\begin{aligned}
k1 &= 8 \text{ X } k2 \\
&= 8 \text{ X } (\ 8 \text{ X } k3\ ) \\
&= 8^2 \text{ X } k3 \\
&= 8^2 \text{ X } (\ 8 \text{ X } k4\ ) \\
&= 8^3 \text{ X } k4 \\
&= \ldots \\
&= 8^{(N-3)} \text{ X } k(N-2) \\
&= 8^{(N-2)} \ (\text{car } k(N-2) = 8)
\end{aligned}
$$

Evaluation de la complexité

Interprétation du résultat

nbre total chemins = $8^{(N-2)}$

$8^{(N-2)} \sim 10^{(N-2)}$

N = nbre total de cases

Ce n’est pas tout à fait correct :

- les 8 déplacements ne sont pas tjrs possibles
- la profondeur d’un chemin n’est pas tjrs N
- N >= 9 sinon le cavalier ne peut pas bouger
- cas le + défavorable

Organisation du code

```
public class Cavalier {

    // Variables globales
    final int TAILLE_ECHEC  = 5 ; // constante
    int[][] damier ;
    int numCoup ;

    void principal () {

        lanceur() ;
    }

    …
}
```

Le tableau à 2 dimensions « damier » mémorise les déplacements du cavalier sur le damier.

La variable « numCoup » comptabilise les déplacements sur le damier.

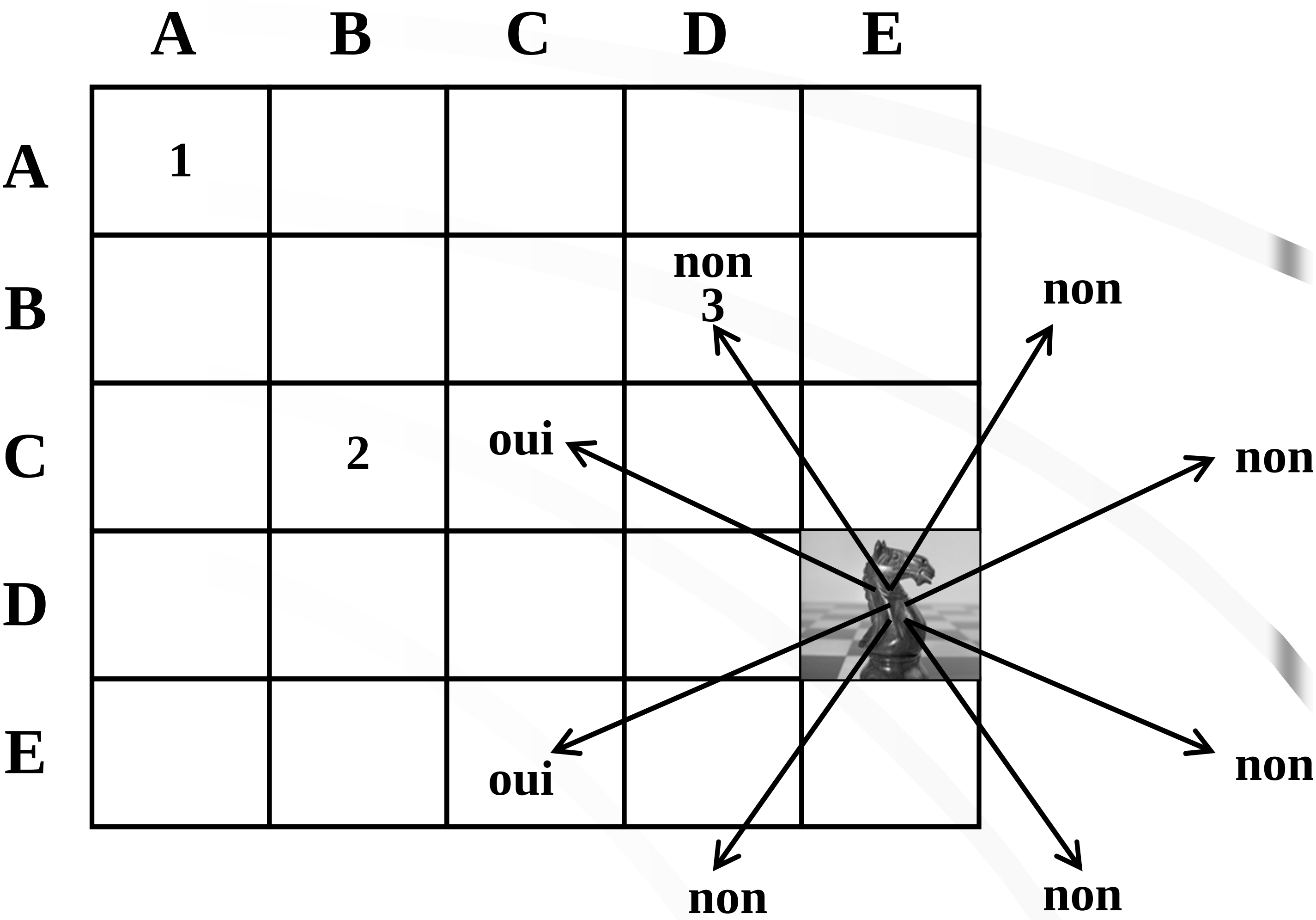
Par convention : « damier[0][0] » désigne la case supérieure gauche.

Organisation du code

```
public class Cavalier {

    // Variables globales
    final int TAILLE_ECHEC  = 5 ; // constante
    int[][] damier ;
    int numCoup ;

    void lanceur () {
        …
        // Choisir une position de départ pour le
        // cavalier
        posX = 0 ;
        posY = 0 ;
        // Premier coup joué en (0, 0)
        numCoup = 1;
        damier [posX, posY] = numCoup;

        // Appeler la fonction récursive de recherche
        // de la solution
        succes = essayer ( posX, posY );

        if ( succes ) {
            afficherDamier ( );
        }
    }
}
```

Organisation du code

Le cavalier étant sur une case valide, quels sont les déplacements possibles ?



Organisation du code

La méthode « donnerSuivants » remplit le tableau « candidats » des 8 déplacements potentiels du cavalier à partir de la case (posX, posY). Le tableau « candidats » contient 8 couples de coordonnées (X, Y).

```
void donnerSuivants (int posX,int posY,int[][] candidats) {

    candidats[0][0] = posX + 2 ;
    candidats[0][1] = posY + 1 ;

    etc.

}
```

Organisation du code

La méthode « estCeValide » renvoie « vrai » si le déplacement du cavalier sur cette nouvelle case (newX, newY) est valide c-à-d :

- que le cavalier ne sort pas du damier
- et que la nouvelle case n'a pas déjà été visitée

```
boolean estCeValide ( int newX, int newY ) {

    boolean ret = true;

    …

    return ret;
}
```