3 Récursivité sur les Données

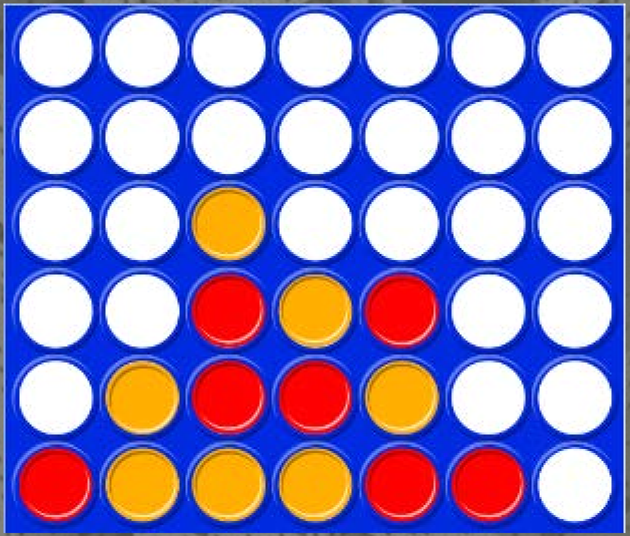
Puissance 4 & Hanoi
Tri Fusion & Rapide

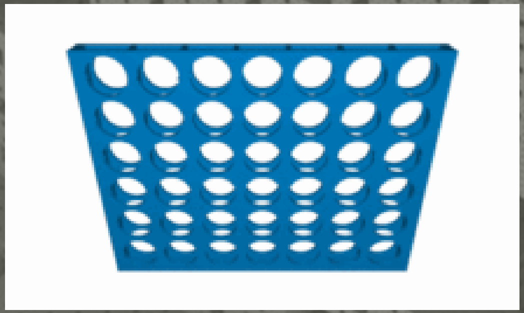
Résoudre un problème en récursif

- **Première question**
  - Identifier la récurrence (lien entre le cas *n* et le cas *n-1*)
  - Réponse = Appels récursifs
- **Deuxième question**
  - Dans quels cas suis-je capable de donner directement le résultat ?
  - Réponse = Cas d'arrêt(s)
- **Dernière question**
  - Est-ce que mon algorithme se termine ?
    - Ai-je prévu tous les cas possibles des paramètres ?

Problème du Puissance4

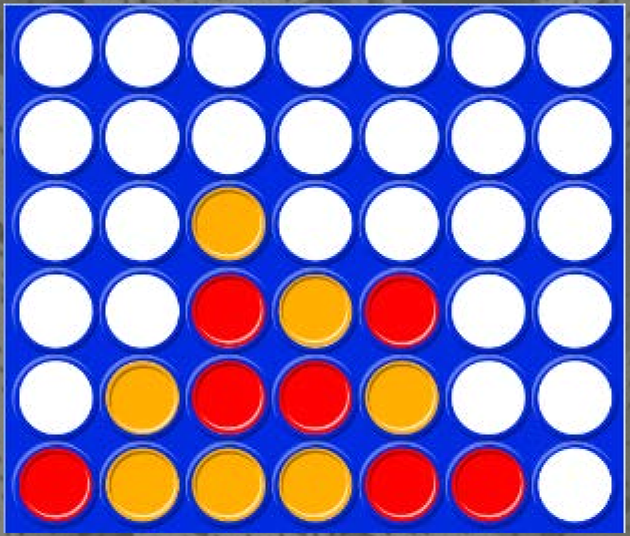
- Déterminer si un joueur à gagner ?
- Pour gagner
  - Aligner 4 pions en vertical, en horizontal ou en diagonale

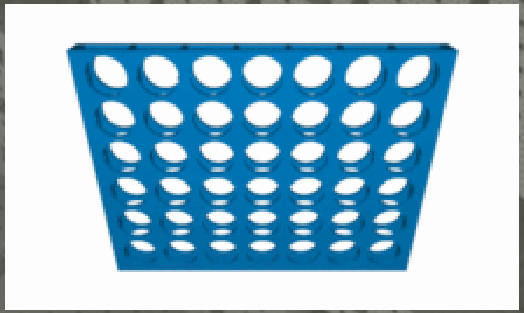




Problème du Puissance4

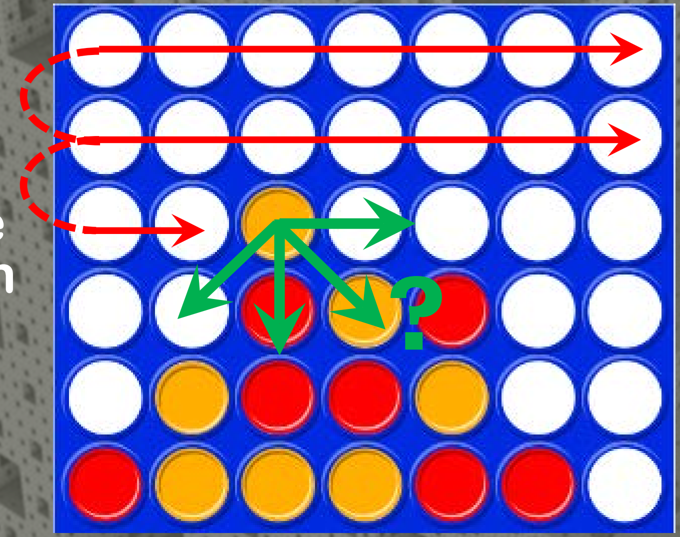
- Déterminer si un joueur à gagner ?
- Pour gagner
  - Aligner 4 pions en vertical, en horizontal ou en diagonale
  - Stratégies (résolues en 1988)
    - Le seul premier coup gagnant est celui dans la colonne centrale
    - Un premier coup dans colonnes adjacentes permet au second joueur d'obtenir une partie nulle
    - Un premier coup dans autres colonnes extérieures permet au second joueur de remporter la victoire





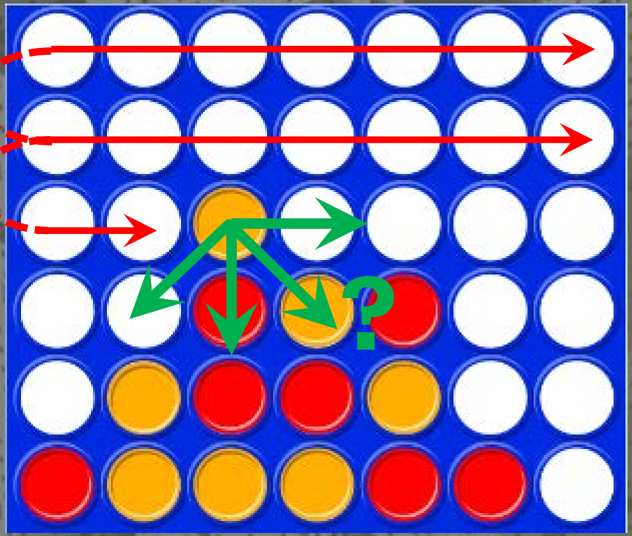
Problème du Puissance4

- Idées de résolution
  - Parcourir le plateau de jeu case après case
  - Pour chaque case, calculer le nb de pions alignés dans chaque direction



Problème du Puissance4

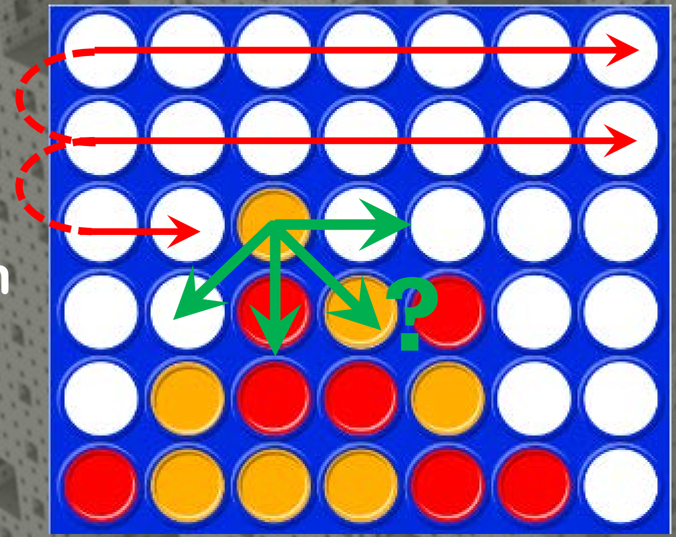
- Idées de résolution
  - Parcourir le plateau de jeu case après case
  - Pour chaque case, calculer le nb de pions alignés dans chaque direction



- Cas d'arrêt
  - Si limites du tableau dépassées ou mauvaise couleur du pion
    $\Rightarrow$ 0 pions alignés

Problème du Puissance4

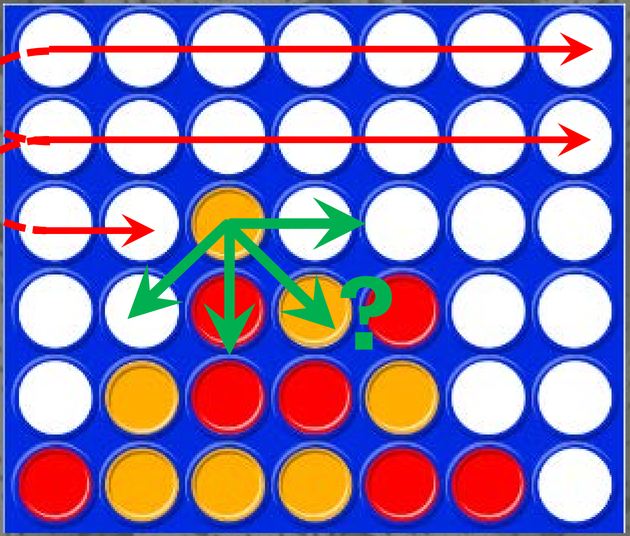
- Idées de résolution
  - Parcourir le plateau de jeu case après case
  - Pour chaque case, calculer le nb de pions alignés dans chaque direction



- Cas d'arrêt
  - Si limites du tableau dépassées ou mauvaise couleur du pion ⇒ 0 pions alignés
- Récurrence
  - Si pion courant est celui du joueur ⇒ pions alignés = 1+ nb de pions alignés sur cases suivantes

Problème du Puissance4

```
Function Gagne(p4 : Plateau, joueur : Entier) : Booleen
   In   : p4, joueur
   Do   : indique si joueur a aligné 4 pions
begin
    foreach case ∈ p4 do
        if case.joueur()=joueur then
            foreach dir ∈ {(0,1),(1,0),(1,1),(1,−1)} do
                if NbPionsDir (p4, joueur, case, dir) ≥ 4 then
                    return True
    return False
```

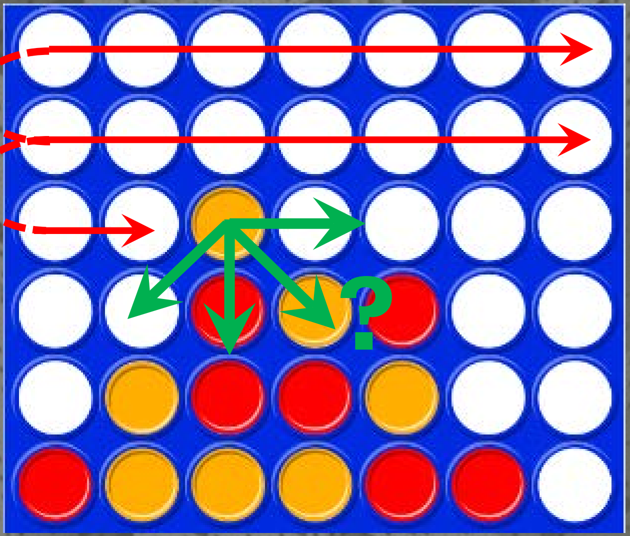


Problème du Puissance4

Function Gagne(p4 : Plateau, joueur : Entier) : Booleen

In : p4, joueur
Do : indique si joueur a aligné 4 pions

```
begin
    foreach case ∈ p4 do
        if case.joueur()=joueur then
            foreach dir ∈ {(0,1),(1,0),(1,1),(1,−1)} do
                if NbPionsDir (p4, joueur, case, dir) ≥ 4 then
                    return True
    return False
```



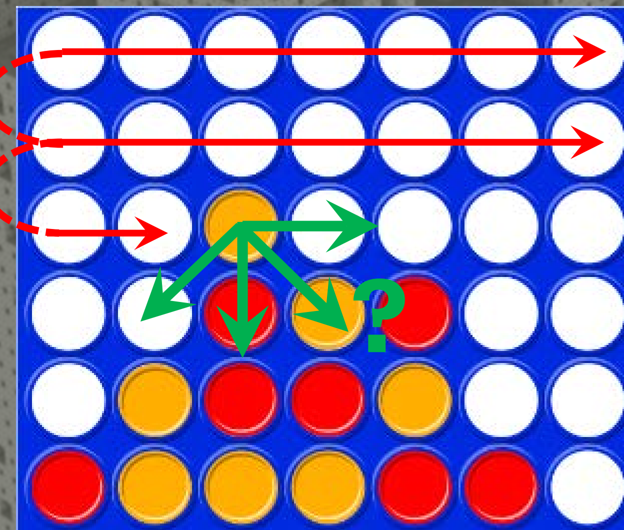
Function NbPionsDir(p4 : Plateau, joueur : Entier, pos : Case, dir : Vecteur) : Entier

In : p4, joueur, pos, dir
Do : compte le nb de pions de joueur alignées dans la direction dir à partir de la case pos

```
begin
    if EstDans (p4, pos) then
        if pos.joueur()=joueur then
            return 1+NbPionsDir (p4, joueur, pos+dir, dir)
    else
        return 0
```

Problème du Puissance4

```
Function Gagne(p4 : Plateau, joueur : Entier) : Booleen
  In   : p4, joueur
  Do   : indique si joueur a aligné 4 pions
begin
  foreach case ∈ p4 do
    if case.joueur()=joueur then
      foreach dir ∈ {(0,1),(1,0),(1,1),(1,−1)} do
        if NbPionsDir (p4, joueur, case, dir) ≥ 4 then
          return True
  return False
```



Terminale ou Non-Terminale ?

```
Function NbPionsDir(p4 : Plateau, joueur : Entier, pos : Case, dir : Vecteur) : Entier
  In   : p4, joueur, pos, dir
  Do   : compte le nb de pions de joueur alignées dans la direction dir à partir de la case pos
begin
  if EstDans (p4, pos) then
    if pos.joueur()=joueur then
      return 1+NbPionsDir (p4, joueur, pos+dir, dir)
  else
    return 0
```

Tour de Hanoï

- **Objectif du jeu**
  - **Déplace *n* disques de tailles différentes de TD vers TA (N° Disque = taille Disque)**

Tour de Hanoï

- **Objectif du jeu**
  - **Déplace *n* disques de tailles différentes de TD vers TA (N° Disque = taille Disque)**
  - **Règles**
    - **Seulement 3 Tours (TD, TI, TA)**
    - **Déplacer un seul disque d_i à la fois**
    - **Déplacer d_1 sur d_2 seulement si $d_1 \leq d_2$**

Tour de Hanoï

Objectif du jeu

- **Déplace *n* disques de tailles différentes de TD vers TA (N° Disque = taille Disque)**
- **Règles**
  - **Seulement 3 Tours (TD, TI, TA)**
  - **Déplacer un seul disque d_i à la fois**
  - **Déplacer d_1 sur d_2 seulement si $d_1 \leq d_2$**
- **Démo**

Tour de Hanoï

Objectif du jeu

- Déplace *n* disques de tailles différentes de TD vers TA (N° Disque = taille Disque)
- Règles
  - Seulement 3 Tours (TD, TI, TA)
  - Déplacer un seul disque d_i à la fois
  - Déplacer d_1 sur d_2 seulement si $d_1 \leq d_2$
- Démo
- Questions
  - Est-ce possible $\forall\ n > 0$?

Tour de Hanoï

Objectif du jeu

- Déplace *n* disques de tailles différentes de TD vers TA (N° Disque = taille Disque)

- Règles
  - Seulement 3 Tours (TD, TI, TA)
  - Déplacer un seul disque d_i à la fois
  - Déplacer d_1 sur d_2 seulement si $d_1 \leq d_2$

- Démo

- Questions
  - Est-ce possible $\forall\ n > 0$?
  - Nb de lignes de l'algorithme ? 5, 10, 20, >20, f(n) ?

1
2
3
4
n

TD TI TA

Tour de Hanoï

- **Récurrence ?**
  - **Pour n=1 $\Rightarrow$ déplace disque 1 de TD $\rightarrow$ TA**

Tour de Hanoï

- Récurrence ?
  - Pour n=1 $\Rightarrow$ déplace disque 1 de TD $\rightarrow$ TA
  - Pour n=2 $\Rightarrow$ on sait faire

```
  TD       TI       TA
Move #0
  _|_      |        |
 __|__     |        |

Move #1
   |       |        |
 __|__    _|_       |

Move #2
   |       |        |
   |      _|_     __|__

Move #3
   |       |       _|_
   |       |      __|__
```

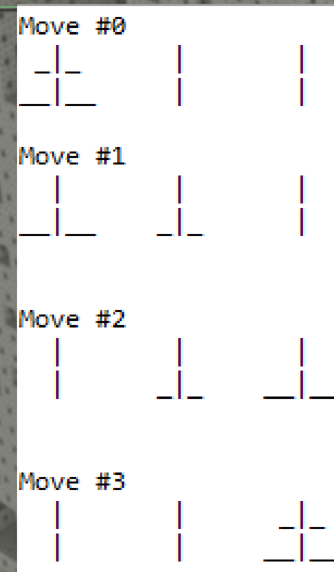
Tour de Hanoï

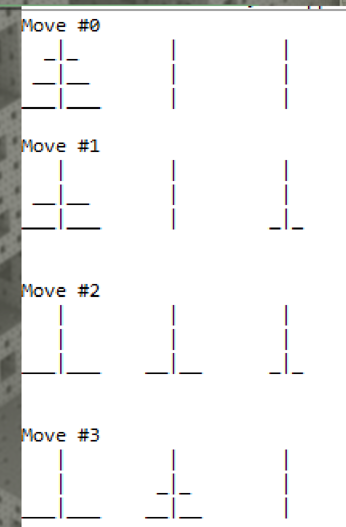
- **Récurrence ?**
  - **Pour n=1 $\Rightarrow$ déplace disque 1 de TD $\rightarrow$ TA**
  - **Pour n=2 $\Rightarrow$ on sait faire**
  - **Pour n=3 ?**

```
  TD      TI      TA
Move #0
  _|_      |       |
 __|__     |       |

Move #1
   |       |       |
 __|__    _|_      |

Move #2
   |       |       |
   |      _|_    __|__

Move #3
   |       |      _|_
   |       |     __|__
```

Tour de Hanoï

TD TI TA TD TI TA



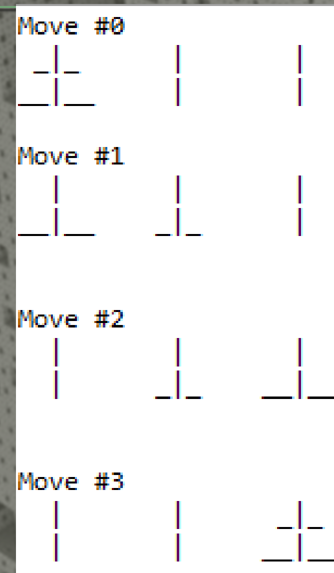


- **Récurrence ?**
  - **Pour n=1 ⇒ déplace disque 1 de TD → TA**
  - **Pour n=2 ⇒ on sait faire**
  - **Pour n=3 ?**

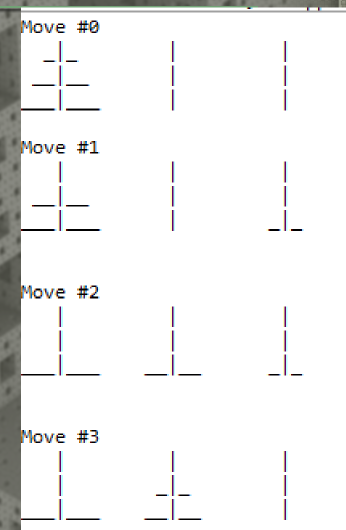
Tour de Hanoï

- Récurrence ?
  - Pour n=1 ⇒ déplace disque 1 de TD → TA
  - Pour n=2 ⇒ on sait faire
  - Pour n=3 ?
    - Déplacer 2 disques de TD → TI

TD TI TA



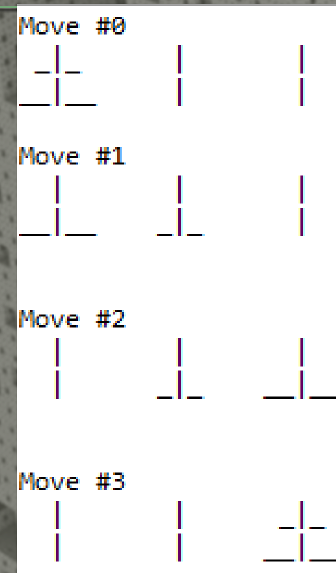
TD TI TA



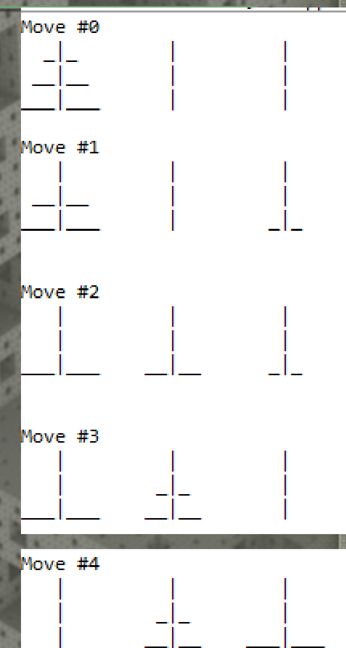
Tour de Hanoï

- **Récurrence ?**
  - Pour n=1 ⇒ déplace disque 1 de TD → TA
  - Pour n=2 ⇒ on sait faire
  - Pour n=3 ?
    - Déplacer 2 disques de TD → TI
    - Déplacer disque 3 de TD → TA

TD TI TA



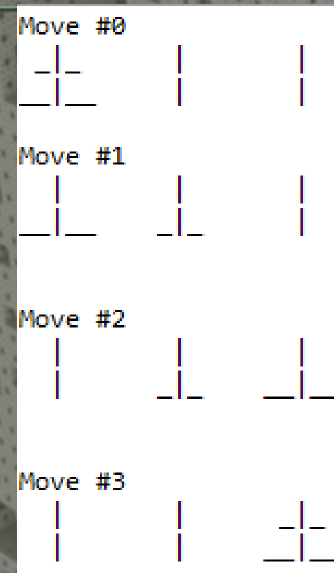
TD TI TA



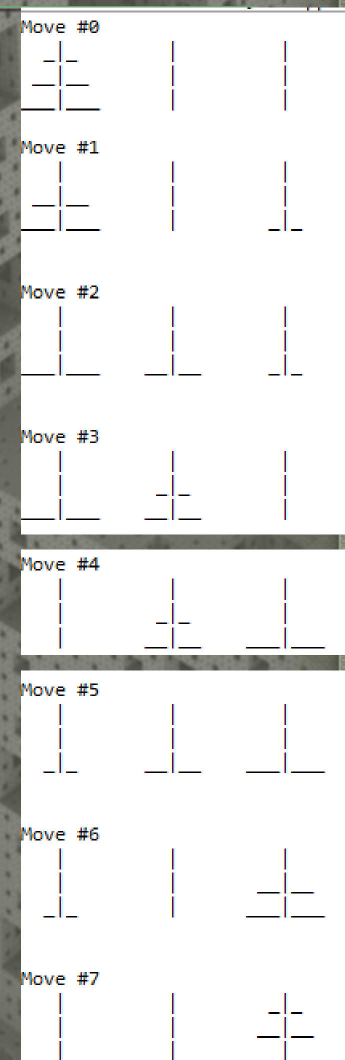
Tour de Hanoï

- Récurrence ?
  - Pour n=1 ⇒ déplace disque 1 de TD → TA
  - Pour n=2 ⇒ on sait faire
  - Pour n=3 ?
    - Déplacer 2 disques de TD → TI
    - Déplacer disque 3 de TD → TA
    - Déplacer 2 disques de TI → TA

TD TI TA

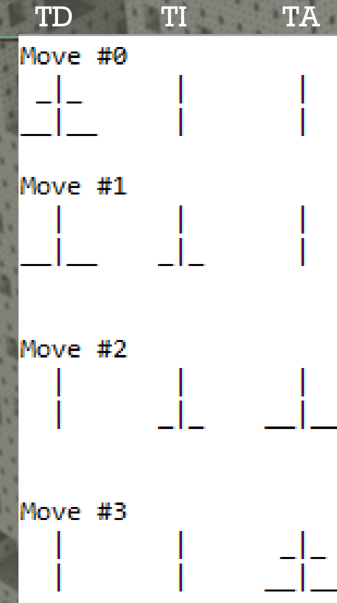


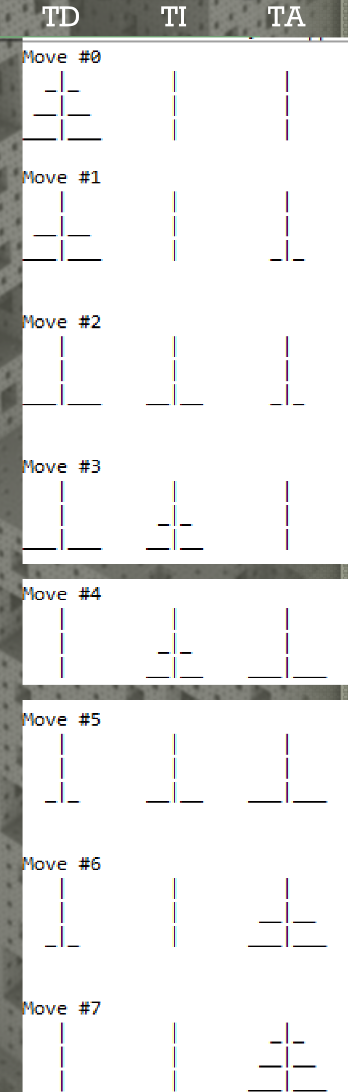
TD TI TA



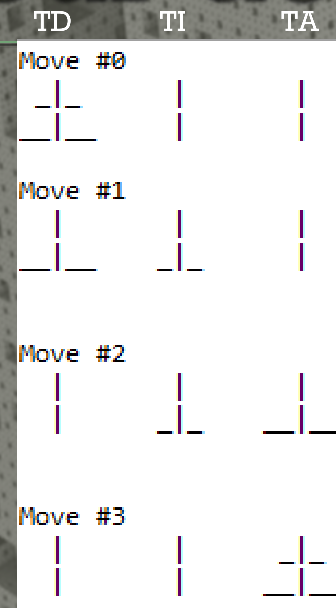
Tour de Hanoï

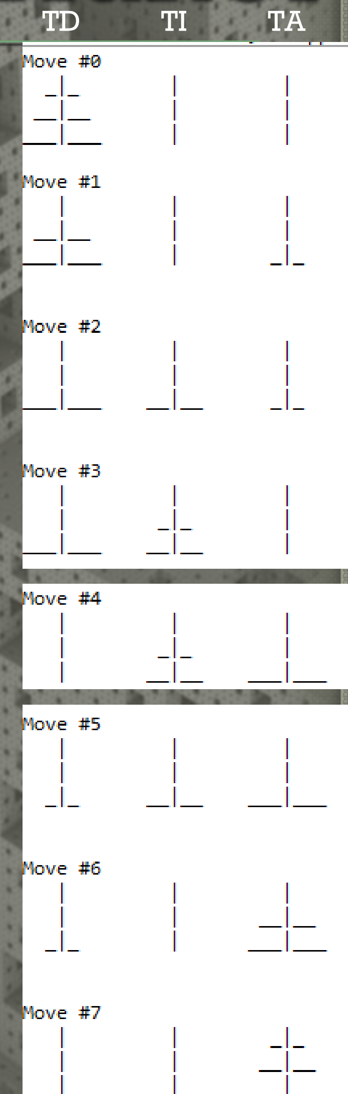
- Récurrence ?
  - Pour n=1 ⇒ déplace disque 1 de TD → TA
  - Pour n=2 ⇒ on sait faire
  - Pour n=3 ?
    - Déplacer 2 disques de TD → TI
      - Hanoi(n=2) de TD vers TI
    - Déplacer disque 3 de TD → TA
    - Déplacer 2 disques de TI → TA





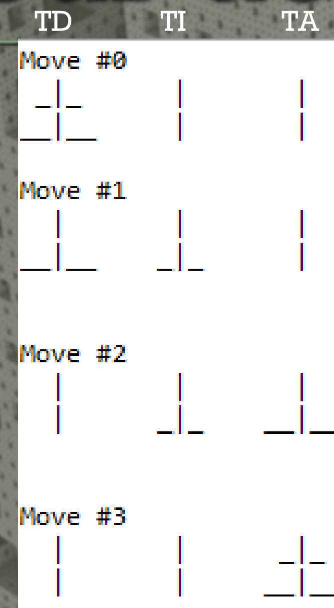
Tour de Hanoï

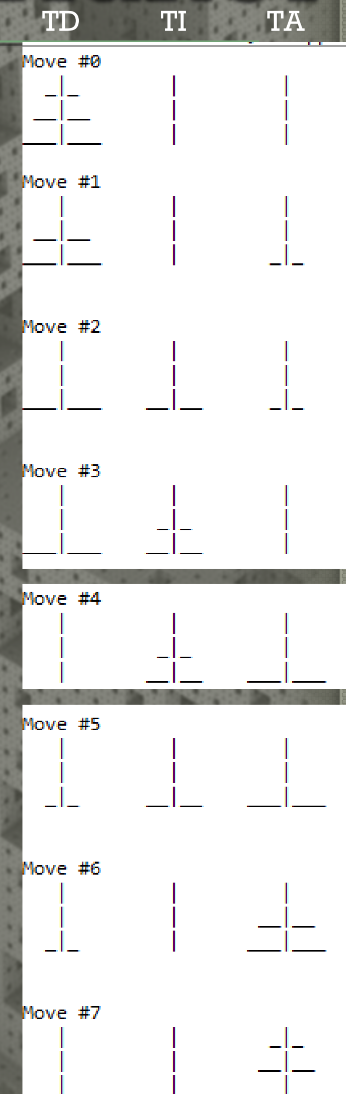




- Récurrence ?
  - Pour n=1 ⇒ déplace disque 1 de TD → TA
  - Pour n=2 ⇒ on sait faire
  - Pour n=3 ?
    - Déplacer 2 disques de TD → TI
      - Hanoi(n=2) de TD vers TI
    - Déplacer disque 3 de TD → TA
    - Déplacer 2 disques de TI → TA
      - Hanoi(n=2) de TI vers TA

Tour de Hanoï

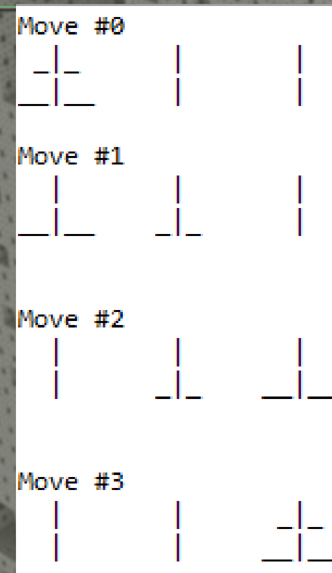


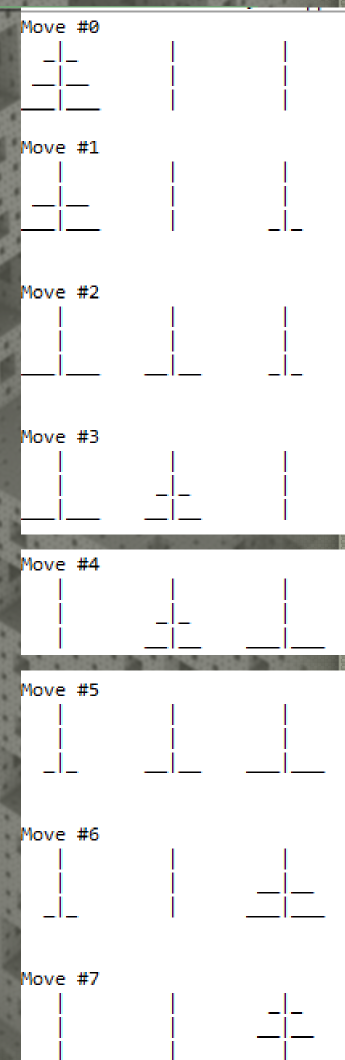


- Récurrence ?
  - Pour n=1 ⇒ déplace disque 1 de TD → TA
  - Pour n=2 ⇒ on sait faire
  - Pour n=3 ?
    - Déplacer 2 disques de TD → TI
      - Hanoi(n=2) de TD vers TI
    - Déplacer disque 3 de TD → TA
    - Déplacer 2 disques de TI → TA
      - Hanoi(n=2) de TI vers TA

⇒ ∀ n > 0, proc. Hanoi (nb disques + 3 tours)

Tour de Hanoï





- **Récurrence ?**
  - Pour n=1 ⇒ déplace disque 1 de TD → TA
  - Pour n=2 ⇒ on sait faire
  - Pour n=3 ?
    - Déplacer 2 disques de TD → TI
      - Hanoi(n=2) de TD vers TI
    - Déplacer disque 3 de TD → TA
    - Déplacer 2 disques de TI → TA
      - Hanoi(n=2) de TI vers TA

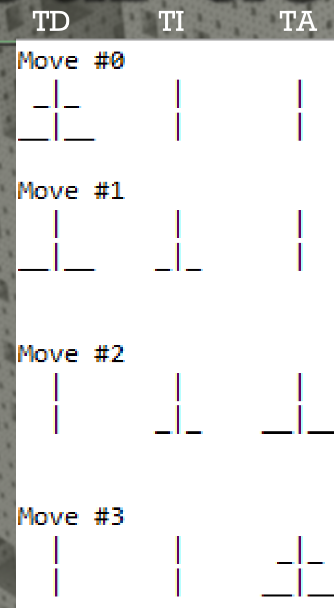
⇒ ∀ n > 0, proc. Hanoi (nb disques + 3 tours)

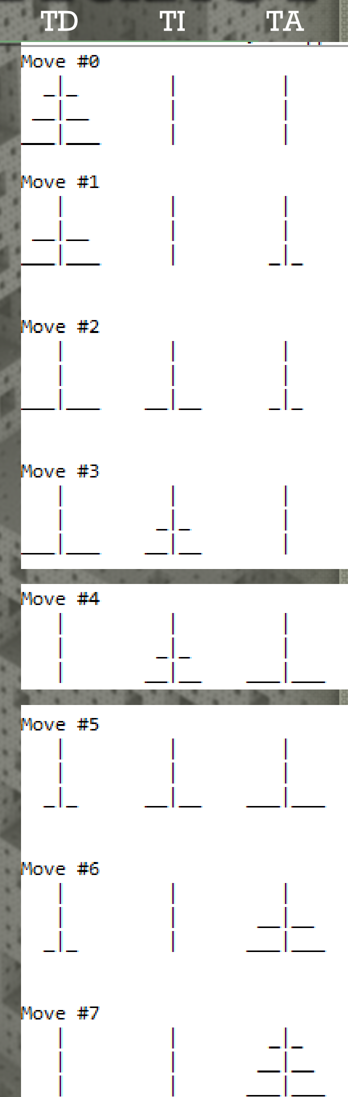
Procedure Hanoi(n, TD, TI, TA : Entier)

In : n=nb de disques, TD=Tour Départ, TA=Tour Arrivée, TI=Tour Intermédiaire
Do : n disques déplacés de TD vers TA en passant par TI

```
begin
    if n > 0 then
        Hanoi(n − 1,TD,TA,TI) // déplace n − 1 disques de TD vers TI par TA
        Deplace (n, TD, TA) // déplace le disque n de TD vers TA
        Hanoi(n − 1,TI,TD,TA) // déplace n − 1 disques de TI vers TA par TD
```

Tour de Hanoï





- **Récurrence ?**
  - Pour n=1 ⇒ déplace disque 1 de TD → TA
  - Pour n=2 ⇒ on sait faire
  - Pour n=3 ?
    - Déplacer 2 disques de TD → TI
      - Hanoi(n=2) de TD vers TI
    - Déplacer disque 3 de TD → TA
    - Déplacer 2 disques de TI → TA
      - Hanoi(n=2) de TI vers TA

⇒ ∀ n > 0, proc. Hanoi (nb disques + 3 tours)

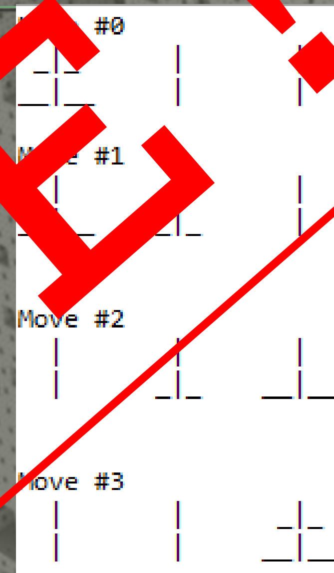
Procedure Hanoi(n, TD, TI, TA : Entier)

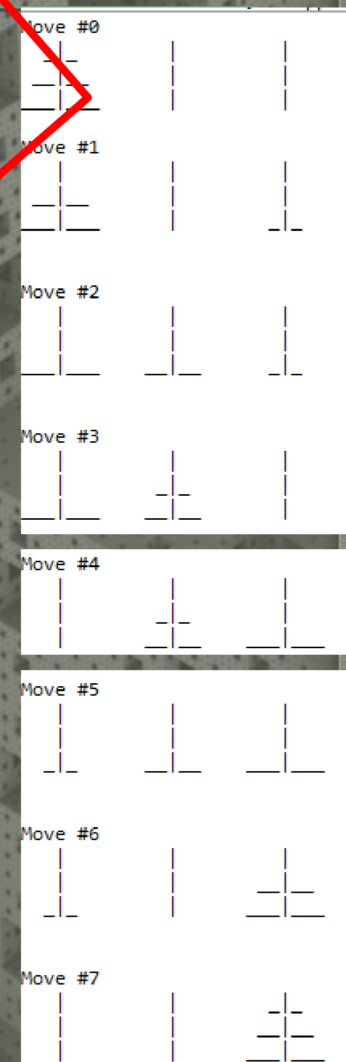
In : n=nb de disques, TD=Tour Départ, TA=Tour Arrivée, TI=Tour Intermédiaire
Do : n disques déplacés de TD vers TA en passant par TI

begin
  if $n > 0$ **then**
    Hanoi($n-1, TD, TA, TI$) `// déplace` $n-1$ `disques de TD vers TI par TA`
    `Deplace` (n, TD, TA) `// déplace le disque` n `de TD vers TA`
    Hanoi($n-1, TI, TD, TA$) `// déplace` $n-1$ `disques de TI vers TA par TD`

- **Hanoi (n+1, TD, TI, TA) ?**

Tour de Hanoï





TRACE !

- **Récurrence ?**
  - Pour n=1 ⇒ déplace disque 1 de TD → TA
  - Pour n=2 ⇒ on sait faire
  - Pour n=3 ?
    - Déplacer 2 disques de TD → TI
      - Hanoi(n=2) de TD vers TI
    - Déplacer disque 3 de TD → TA
    - Déplacer 2 disques de TI → TA
      - Hanoi(n=2) de TI vers TA

⇒ ∀ n > 0, proc. Hanoi(nb disques + 3 tours)

```
Procedure Hanoi(n, TD, TI, TA: Entier)
In   : n=nb de disques, TD=Tour Départ, TA=Tour Arrivée, TI=Tour Intermédiaire
Do   : n disques déplacés de TD vers TA en passant par TI
begin
    if n > 0 then
        Hanoi(n − 1, TD, TA, TI) // déplace n − 1 disques de TD vers TI par TA
        Deplace (n, TD, TA) // déplace le disque n de TD vers TA
        Hanoi(n − 1, TI, TD, TA) // déplace n − 1 disques de TI vers TA par TD
```

- Hanoi (n+1, TD, TI, TA) ?

Algorithmes & Structures de données

- **Algorithme**
  - Lié à la SdD qu'il utilise / manipule

Algorithmes & Structures de données

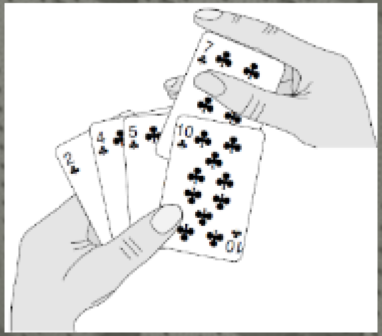
- Algorithme
  - Lié à la SdD qu'il utilise / manipule
- Objectif
  - Trouver 1 algo efficace sur 1 SdD la moins coûteuse

    ⇒ Coût / efficacité d'1 algo abordé en M415

Algorithmes & Structures de données

- Algorithme
  - Lié à la SdD qu'il utilise / manipule
- Objectif
  - Trouver 1 algo efficace sur 1 SdD la moins coûteuse

    ⇒ Coût / efficacité d'1 algo abordé en M415
- Exemple
  - Recherche d'un élément dans 1 tableau (1 carte dans un jeu)
    - Il faut parcourir tout le tableau pour savoir si l'élément est dedans ou non → pas optimal

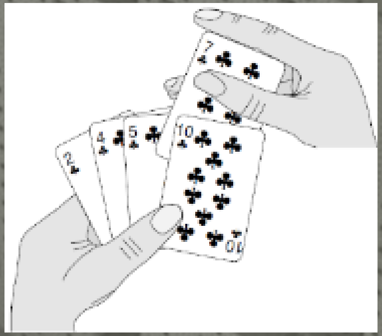


Algorithmes & Structures de données

- Algorithme
  - Lié à la SdD qu'il utilise / manipule
- Objectif
  - Trouver 1 algo efficace sur 1 SdD la moins coûteuse

    ⇒ Coût / efficacité d'1 algo abordé en M415

- Exemple
  - Recherche d'un élément dans 1 tableau (1 carte dans un jeu)
    - Il faut parcourir tout le tableau pour savoir si l'élément est dedans ou non → pas optimal
    - Mais si tableau trié alors beaucoup moins de temps



Recherche d'un élément dans un tableau (récursif)

Function rechercheElement(t : Tableau, min : Entier, e : Element) : Entier

In : t, min, e

Do : Recherche l'indice de *e* dans *t* (supposé non trié) à partir de l'indice *min* renvoie -1 si pas trouvé

Recherche d'un élément dans un tableau (récursif)

Function rechercheElement(t : Tableau, min : Entier, e : Element) : Entier

In : t, min, e

Do : Recherche l'indice de *e* dans *t* (supposé non trié) à partir de l'indice *min* renvoie -1 si pas trouvé

```
begin
    if min ≥ 0 and min < t.taille then
        if t[min] = e then return min
        else return rechercheElement(t, min+1, e)
    else
        return −1
```

Recherche d'un élément dans un tableau (récursif)

```
Function rechercheElement(t : Tableau, min : Entier, e : Element) : Entier
  In   : t, min, e
  Do   : Recherche l'indice de e dans t (supposé non trié) à partir de l'indice min
         renvoie -1 si pas trouvé
begin
  if min ≥ 0 and min < t.taille then
    if t[min] = e then return min
    else return rechercheElement(t, min+1, e)
  else
    return −1
```

Appel initial avec
rechercheElement([2,5,10…],0,6)

Recherche d'un élément dans un tableau (récursif)

Function rechercheElement(t : Tableau, min : Entier, e : Element) : Entier

In : t, min, e
Do : Recherche l'indice de e dans t (supposé non trié) à partir de l'indice min
renvoie -1 si pas trouvé

begin
  if $min \geq 0$ **and** $min < t.taille$ **then**
    if $t[min] = e$ **then return** min
    else return rechercheElement(t, min+1, e)
  else
    return -1

Appel initial avec
rechercheElement([2,5,10…],0,6)

Function rechercheDico(t : Tableau, min : Entier, max : Entier, e : Element) : Entier

In : t, min, max, e
Do : Recherche l'indice de e dans t (supposé **trié**) entre les indices min et max
renvoie -1 si pas trouvé

begin
  if $min = max$ **and** $min \in [0, t.taille[$ **then**
    if $t[min] = e$ **then return** min
    else return -1
  $mid \leftarrow \frac{(min+max)}{2}$
  if $t[mid] < e$ **then return** RechercheDico(t, mid+1, max, e)
  else return RechercheDico(t, min, mid, e)

Recherche d'un élément dans un tableau (récursif)

Function rechercheElement(t : Tableau, min : Entier, e : Element) : Entier

In : t, min, e
Do : Recherche l'indice de e dans t (supposé non trié) à partir de l'indice min renvoie -1 si pas trouvé

```
begin
    if min ≥ 0 and min < t.taille then
        if t[min] = e then return min
        else return rechercheElement(t, min+1, e)
    else
        return −1
```

Appel initial avec rechercheElement([2,5,10…],0,6)

Function rechercheDico(t : Tableau, min : Entier, max : Entier, e : Element) : Entier

In : t, min, max, e
Do : Recherche l'indice de e dans t (supposé **trié**) entre les indices min et max renvoie -1 si pas trouvé

```
begin
    if min = max and min ∈ [0, t.taille[ then
        if t[min] = e then return min
        else return −1
    mid ← (min+max)/2
    if t[mid] < e then return RechercheDico(t, mid+1, max, e)
    else return RechercheDico(t, min, mid, e)
```

Appel initial avec rechercheDico([2,5,10…],0,taille(t)-1,6)

Tris récursifs

- Paradigme « Diviser pour Régner »
  - On divise un problème de grande taille en plusieurs (au moins deux) sous-problèmes analogues

Tris récursifs

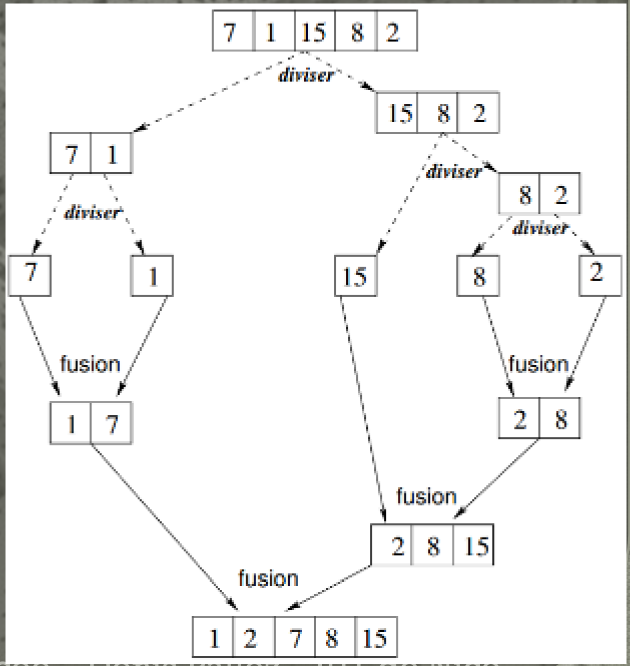
- Paradigme « Diviser pour Régner »
  - On divise un problème de grande taille en plusieurs (au moins deux) sous-problèmes analogues
  - Récursivité sur les données
    - Séparation des données en 2 parties, résolution du problème sur chacune des parties et combinaison des résultats
    - Tri par fusion (MergeSort) = Fusion de sous-parties

Tris récursifs

- **Paradigme « Diviser pour Régner »**
  - On divise un problème de grande taille en plusieurs (au moins deux) sous-problèmes analogues
  - Récursivité sur les données
    - Séparation des données en 2 parties, résolution du problème sur chacune des parties et combinaison des résultats
    - Tri par fusion (MergeSort) = **Fusion** de sous-parties
  - Récursivité sur les résultats
    - Découpage intelligent des données, puis résolution des problèmes pour que la combinaison se fasse plus facilement Tri rapide (QuickSort) = **Partitionnement** de sous-parties

Tri par fusion

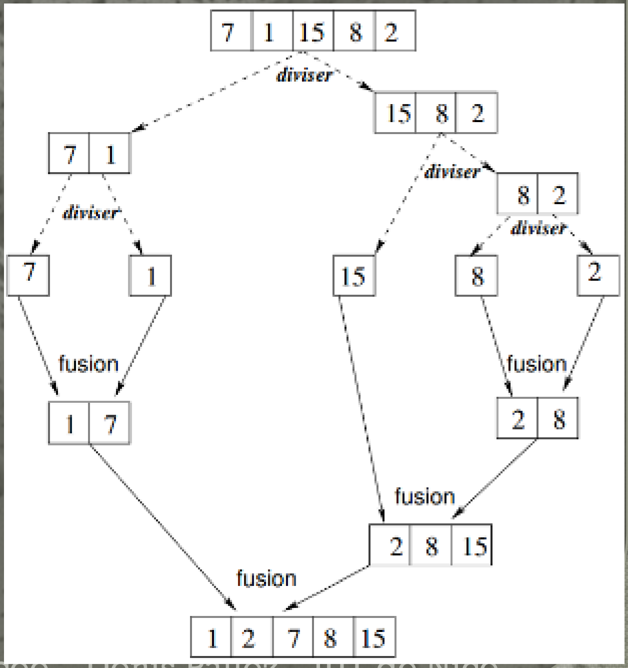
- Soit V un ensemble de *n* valeurs non triées
- 3 étapes
  - Diviser V en 2 parties V_1, V_2 de dimension n/2
  - Appliquer le tri par fusion sur V_1 et sur V_2
  - Fusionner V_1 et V_2 (triés) dans V



Tri par fusion

- Soit V un ensemble de *n* valeurs non triées
- 3 étapes
  - Diviser V en 2 parties V_1, V_2 de dimension n/2
  - Appliquer le tri par fusion sur V_1 et sur V_2
  - Fusionner V_1 et V_2 (triés) dans V

```
Procedure triFusion(t : Tableau, min : Entier, max : Entier)
  In     : min, max
  In/Out : t
  Do     : tri le tableau t entre les indices min et max
begin
    if min ≠ max then
        mid ← (min+max)/2
        triFusion(t, min, mid)
        triFusion(t, mid+1, max)
        Tableau tmp [max − min + 1]
        Fusion(t[min, mid], t[mid + 1, max], tmp)
        ∀i ∈ [0, max − min],   t[min + i] ← tmp[i]
```



Tri par fusion

- Fusion de 2 tableaux triés

Tri par fusion

- **Fusion de 2 tableaux triés**
  - Comparer le plus petit élément de V1 avec le plus petit élément de V2, et insérer le plus petit des deux dans V

Tri par fusion

- Fusion de 2 tableaux triés
  - Comparer le plus petit élément de V1 avec le plus petit élément de V2, et insérer le plus petit des deux dans V
  - Recopier le reste du tableau

Tri par fusion

- **Fusion de 2 tableaux triés**
  - Comparer le plus petit élément de V1 avec le plus petit élément de V2, et insérer le plus petit des deux dans V
  - Recopier le reste du tableau

```
Procedure Fusion(t1 : Tableau, t2 : Tableau, t : Tableau)
In      : t1, t2
In/Out: t
Do      : fusionne 2 tableaux triés t1 et t2 dans le tableau t
begin
    i1, i2, i ← 0
    // copie min(t1[i],t2[i]) dans t tant que pas fin(t1) ou fin(t2)
    while i1 < t1.taille and i2 < t2.taille do
        if t1[i1] < t2[i2] then
            t[i]=t1[i1]
            i++; i1++
        else
            t[i]=t2[i2]
            i++; i2++
    // recopie la fin du tableau t1 ou t2 mais lequel ?
    while i1 < t1.taille do t[i]=t1[i1]; i++ ; i1++
    while i2 < t2.taille do t[i]=t2[i2]; i++ ; i2++
```

Tri par fusion

- Fusion de 2 tableaux triés
  - Comparer le plus petit élément de V1 avec le plus petit élément de V2, et insérer le plus petit des deux dans V
  - Recopier le reste du tableau
- Beaucoup de création de tableaux et de copies

```
Procedure Fusion(t1 : Tableau, t2 : Tableau, t : Tableau)
 In      : t1, t2
 In/Out: t
 Do      : fusionne 2 tableaux triés t1 et t2 dans le tableau t
begin
    i1, i2, i ← 0
    // copie min(t1[i],t2[i]) dans t tant que pas fin(t1) ou fin(t2)
    while i1 < t1.taille and i2 < t2.taille do
        if t1[i1] < t2[i2] then
            t[i]=t1[i1]
            i++; i1++
        else
            t[i]=t2[i2]
            i++; i2++
    // recopie la fin du tableau t1 ou t2 mais lequel ?
    while i1 < t1.taille do t[i]=t1[i1]; i++ ; i1++
    while i2 < t2.taille do t[i]=t2[i2]; i++ ; i2++
```

Tri par fusion

- Fusion de 2 tableaux triés
  - Comparer le plus petit élément de V1 avec le plus petit élément de V2, et insérer le plus petit des deux dans V
  - Recopier le reste du tableau
- Beaucoup de création de tableaux et de copies
- AlgoRithmics https://www.youtube.com/watch?v=XaqR3G_NVoo

```
Procedure Fusion(t1 : Tableau, t2 : Tableau, t : Tableau)
  In      : t1, t2
  In/Out: t
  Do      : fusionne 2 tableaux triés t1 et t2 dans le tableau t
begin
   i1, i2, i ← 0
   // copie min(t1[i],t2[i]) dans t tant que pas fin(t1) ou fin(t2)
   while i1 < t1.taille and i2 < t2.taille do
      if t1[i1] < t2[i2] then
         t[i]=t1[i1]
         i++; i1++
      else
         t[i]=t2[i2]
         i++; i2++
   // recopie la fin du tableau t1 ou t2 mais lequel ?
   while i1 < t1.taille do t[i]=t1[i1]; i++ ; i1++
   while i2 < t2.taille do t[i]=t2[i2]; i++ ; i2++
```

Récursivité sur résultats

- **Séparer les éléments d'un tableau en 2 pour que si on trie chaque partie, on obtienne un résultat trié**
- **Exemple**
  - **Partitionnement = Nb pair au début, impair à la fin et tri de chaque partie indépendamment**

  ⇒Tri rapide (QuickSort)

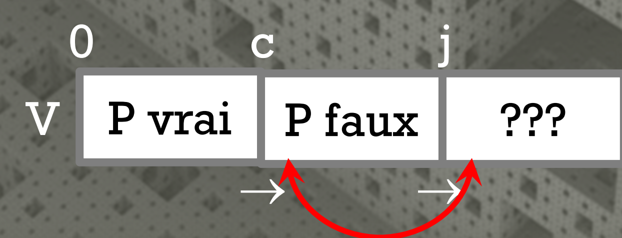
AlgoRithmics
https://www.youtube.com/watch?v=ywWBy6J5gz8

Partitionner

- Soit V un ensemble
  P un prédicat sur les éléments de V
  Ex : « est pair »

- PV={e $\in$V | P(e) est vraie}
  PNV={e $\in$V | P(e) est faux}

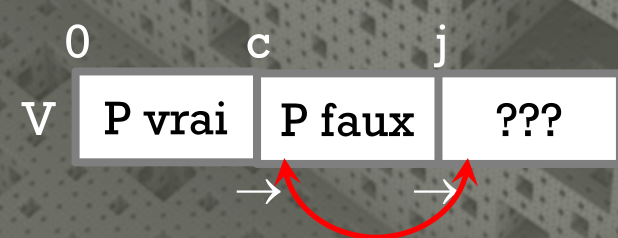
- V = PV $\cup$ PNV;
  PV $\cap$ PNV = $\varnothing$
  (PV, PNV) est appelée une partition de V



Partitionner

- Soit V un ensemble
  P un prédicat sur les éléments de V
  Ex : « est pair »
- PV={e $\in$V | P(e) est vraie}
  PNV={e $\in$V | P(e) est faux}
- V = PV $\cup$ PNV;
  PV $\cap$ PNV = $\emptyset$
  (PV, PNV) est appelée une partition de V

```
Function Partitionner(t : Tableau, p : Predicat) : Entier
  In     : p
  In/Out : t
  Do     : modifie t / éléments qui vérifient p ∈ aux indices [0...c]
           et éléments qui vérifient pas p ∈ [c + 1, t.taille[
begin
    c ← 0
    while c < t.taille and p(t[c]) do c ← c + 1
    for j = c + 1 to t.taille − 1 do
        if p(t[j]) then
            Echanger (t[c], t[j])
            c ← c + 1
    return c
```



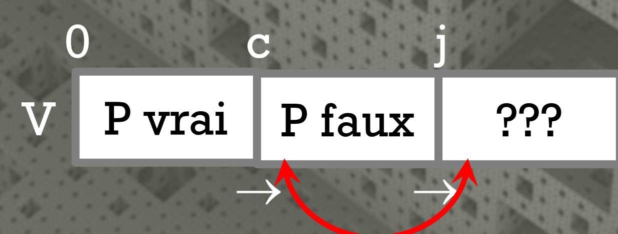
Partitionner

- **Problème**
  - Fait bcp de déplacements
  - Au pire des cas, fait *taille* échanges alors qu'un seul serait nécessaire
  - Exemple
    - Si P(t[0]) est faux et P(t[i]) est vrai
      Alors décale tout le tableau alors qu'il suffirait d'échanger le premier et le dernier élément

Function Partitionner(t : Tableau, p : Predicat) : Entier

In : p
In/Out: t
Do : modifie t / éléments qui **vérifient** p $\in$ aux indices [0...c]
et éléments qui **vérifient pas** p $\in [c+1, t.taille[$

begin
  $c \leftarrow 0$
  while $c < t.taille$ **and** $p(t[c])$ **do** $c \leftarrow c+1$
  for $j = c+1$ **to** $t.taille - 1$ **do**
    if $p(t[j])$ **then**
      Echanger $(t[c], t[j])$
      $c \leftarrow c+1$
  return c



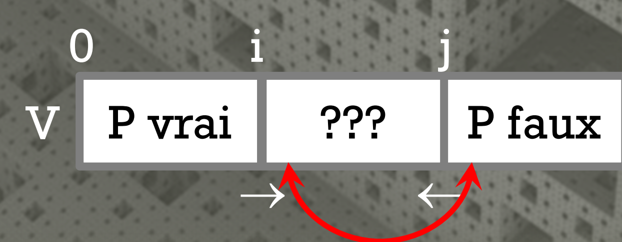
Partitionner

- **Variante**

Function PartitionnerV2(t : Tableau, p : Predicat) : Entier

In : p
In/Out: t
Do : modifie t / éléments qui **vérifient p** $\in$ aux indices [0...c]
et éléments qui **vérifient pas p** $\in [c+1, t.taille[$

```
begin
    i ← 0; j ← t.taille − 1
    while i < j do
        while p(t[i]) do i ← i + 1
        while ¬p(t[j]) do j ← j − 1
        if i < j then
            Echanger(t[i], t[j])
            i ← i + 1; j ← j − 1
    return i
```



Partitionner

- **Variante**
  - **Est-ce que cet algorithme fonctionne dans tous les cas ?**

```
Function PartitionnerV2(t : Tableau, p : Predicat) : Entier
  In     : p
  In/Out : t
  Do     : modifie t / éléments qui vérifient p ∈ aux indices [0...c]
           et éléments qui vérifient pas p ∈ [c + 1, t.taille[
begin
    i ← 0; j ← t.taille − 1
    while i < j do
        while p(t[i]) do i ← i + 1
        while ¬p(t[j]) do j ← j − 1
        if i < j then
            Echanger(t[i], t[j])
            i ← i + 1; j ← j − 1
    return i
```

	0	i	j
V	P vrai	???	P faux

Partitionner

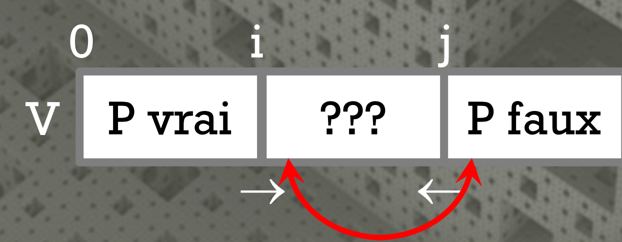
- **Variante**
  - Est-ce que cet algorithme fonctionne dans tous les cas ?

  - Essayer avec t contenant
    1. Que des éléments qui vérifie P ?
    2. Aucun élément vérifie P ?

Function PartitionnerV2(t : Tableau, p : Predicat) : Entier

In : p
In/Out: t
Do : modifie t / éléments qui **vérifient p** $\in$ aux indices [0...c] et éléments qui **vérifient pas p** $\in [c+1, t.taille[$

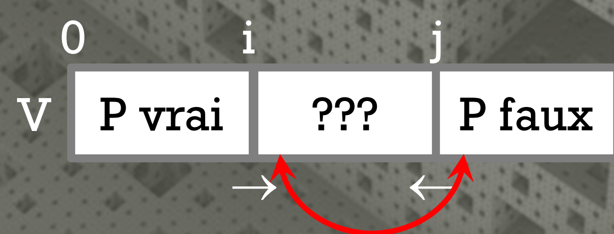
```
begin
    i ← 0; j ← t.taille − 1
    while i < j do
        while p(t[i]) do i ← i + 1
        while ¬p(t[j]) do j ← j − 1
        if i < j then
            Echanger (t[i], t[j])
            i ← i + 1; j ← j − 1
    return i
```



Partitionner

- **Variante**
  - Est-ce que cet algorithme fonctionne dans tous les cas ?
  - Essayer avec t contenant
    1. Que des éléments qui vérifie P ?
    2. Aucun élément vérifie P ?
  - Résultats
    1. while ((P(t[i])) {i++;} fait sortir i
    2. while (not(P(t[j])) {j--;} fait sortir j

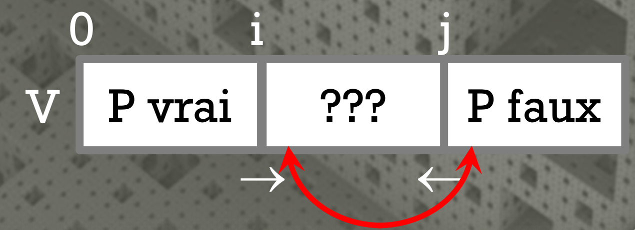
```
Function PartitionnerV2(t : Tableau, p : Predicat) : Entier
  In      : p
  In/Out: t
  Do      : modifie t / éléments qui vérifient p ∈ aux indices [0...c]
            et éléments qui vérifient pas p ∈ [c + 1, t.taille[
begin
    i ← 0; j ← t.taille − 1
    while i < j do
        while p(t[i]) do i ← i + 1
        while ¬p(t[j]) do j ← j − 1
        if i < j then
            Echanger(t[i], t[j])
            i ← i + 1; j ← j − 1
    return i
```



Partitionner

- **Variante**
  - Est-ce que cet algorithme fonctionne dans tous les cas ?
  - Essayer avec t contenant
    1. Que des éléments qui vérifie P ?
    2. Aucun élément vérifie P ?
  - Résultats
    1. while ((P(t[i])) {i++;} fait sortir i
    2. while (not(P(t[j])) {j--;} fait sortir j

```
Function PartitionnerV2(t : Tableau, p : Predicat) : Entier
  In     : p
  In/Out: t
  Do     : modifie t / éléments qui vérifient p ∈ aux indices [0...c]
           et éléments qui vérifient pas p ∈ [c + 1, t.taille[
begin
  i ← 0; j ← t.taille − 1
  while i < j do
    while p(t[i]) do i ← i + 1
    while ¬p(t[j]) do j ← j − 1
    if i < j then
      Echanger (t[i], t[j])
      i ← i + 1; j ← j − 1
  return i
```



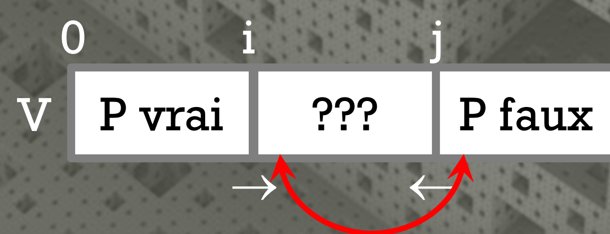
Partitionner

- **Variante OK**

Function PartitionnerV2OK(t : Tableau, p : Predicat) : Entier

In : p
In/Out: t
Do : modifie t / éléments qui **vérifient** **p** $\in$ aux indices $[0...c]$ et éléments qui **vérifient pas p** $\in [c+1, t.taille[$

```
begin
    i ← 0; j ← t.taille − 1
    while i < t.taille and p(t[i]) do i ← i + 1
    while j ≥ 0 and ¬p(t[j]) do j ← j − 1
    while i < j do
        while p(t[i]) do i ← i + 1
        while ¬p(t[j]) do j ← j − 1
        if i < j then
            Echanger(t[i], t[j])
            i ← i + 1; j ← j − 1
    return i
```



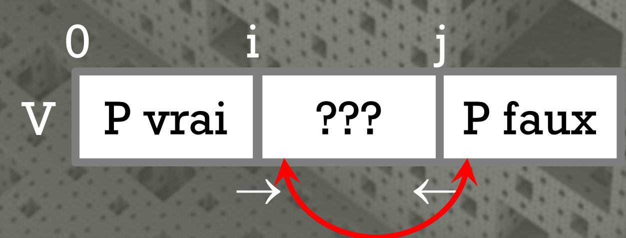
Partitionner

Variante OK

algo + simple, - efficace
algo + compliqué, + efficace
⇒ **Etude complexité = Sem 4 !**

```
Function PartitionnerV2OK(t : Tableau, p : Predicat) : Entier
  In     : p
  In/Out: t
  Do     : modifie t / éléments qui vérifient p ∈ aux indices [0...c]
           et éléments qui vérifient pas p ∈ [c + 1, t.taille[
begin
    i ← 0; j ← t.taille − 1
    while i < t.taille and p(t[i]) do i ← i + 1
    while j ≥ 0 and ¬p(t[j]) do j ← j − 1
    while i < j do
        while p(t[i]) do i ← i + 1
        while ¬p(t[j]) do j ← j − 1
        if i < j then
            Echanger(t[i], t[j])
            i ← i + 1; j ← j − 1
    return i
```

```
Function Partitionner(t : Tableau, p : Predicat) : Entier
  In     : p
  In/Out: t
  Do     : modifie t / éléments qui vérifient p ∈ aux indices [0...c]
           et éléments qui vérifient pas p ∈ [c + 1, t.taille[
begin
    c ← 0
    while c < t.taille and p(t[c]) do c ← c + 1
    for j = c + 1 to t.taille − 1 do
        if p(t[j]) then
            Echanger(t[c], t[j])
            c ← c + 1
    return c
```



Tri rapide

- Soit V un ensemble de *n* données non triées
- 3 étapes
  - Choisir un élément x (appelé **pivot**) et diviser V en 3 parties P, E, G
    - P contient les éléments plus petit que x
    - E contient les éléments égaux à x
    - G contient les éléments plus grand que x

    $\Rightarrow$ En fait, chaque élément y de V est comparé au pivot et inséré dans P, E ou G
  - Trier récursivement P et G
  - Fusionner L, E, G

Tri rapide

- Comment choisir le pivot
  - Stratégies
    - Choisir l'élément au milieu du tableau
    - Choisir un élément au hasard dans le tableau
    - …
  - Les pires des cas
    - si pivot est le plus grand élément et donc est placé à la fin après partitionnement
    - si pivot est le plus petit élément et donc est placé au début après partitionnement

Tri rapide

```
Procedure triRapide(t : Tableau, min : Entier, max : Entier)
  In     : min, max
  In/Out : t
  Do     : tri le tableau t entre indices min et max
           suivant un certain prédicat à définir
begin
    pivot ← SelectPivot(t,min,max)
    [i, j] ← PartitionnerInfPivot(t, min, max, pivot) // prédicat = "<pivot"
    if min < j then triRapide(t,min,j)
    if i < max then triRapide(t,i,max)
```

Tri rapide

Procedure triRapide(t : Tableau, min : Entier, max : Entier)

In : min, max
In/Out: t
Do : tri le tableau t entre indices min et max suivant un certain **prédicat** à définir

```
begin
    pivot ← SelectPivot(t,min,max)
    [i, j] ← PartitionnerInfPivot(t, min, max, pivot) // prédicat = "<pivot"
    if min < j then triRapide(t,min,j)
    if i < max then triRapide(t,i,max)
```

Function PartitionnerV2OK(t : Tableau, p : Predicat) : Entier

In : p
In/Out: t
Do : modifie t / éléments qui **vérifient p** $\in$ aux indices [0...c] et éléments qui **vérifient pas p** $\in [c+1, t.taille[$

```
begin
    i ← 0; j ← t.taille − 1
    while i < t.taille and p(t[i]) do i ← i + 1
    while j ≥ 0 and ¬p(t[j]) do j ← j − 1
    while i < j do
        while p(t[i]) do i ← i + 1
        while ¬p(t[j]) do j ← j − 1
        if i < j then
            Echanger(t[i], t[j])
            i ← i + 1; j ← j − 1
    return i
```

Prédicat = < pivot
ajouter min & max
en paramètres

Tri rapide

Procedure triRapide(t : Tableau, min : Entier, max : Entier)

In : min, max
In/Out: t
Do : tri le tableau t entre indices min et max
suivant un certain **prédicat** à définir

```
begin
    pivot ← SelectPivot(t,min,max)
    [i, j] ← PartitionnerInfPivot(t, min, max, pivot) // prédicat = "<pivot"
    if min < j then triRapide(t,min,j)
    if i < max then triRapide(t,i,max)
```

Function partitionnerInfPivot(t : Tableau, min, Entier, max : Entier, pivot : Entier) : Entiers

In : pivot, min, max
In/Out: t
Do : Structure t en 3 parties (plus petits, égaux ou supérieurs à $pivot$) entre min et max.
i, j représentent les indices qui délimitent les parties

```
begin
    i ← min; j ← max
    while i < max and t[i] < pivot do i ← i + 1
    while j ≥ min and t[j] > pivot do j ← j − 1
    while i < j do
        while t[i] < pivot do i ← i + 1
        while t[j] > pivot do j ← j − 1
        if i < j then
            Echanger(t[i], t[j])
            i ← i + 1; j ← j − 1
    return [i, j]
```

Sources

- Florent Hivert, Algorithmique
- Elise Bonzon, Algo. et structures, Récursion
- Damien Massé, Algo. récursifs : Applications
- Frédéric Fürst, Récursivité