

2.5. Minimax





CPE10040H9





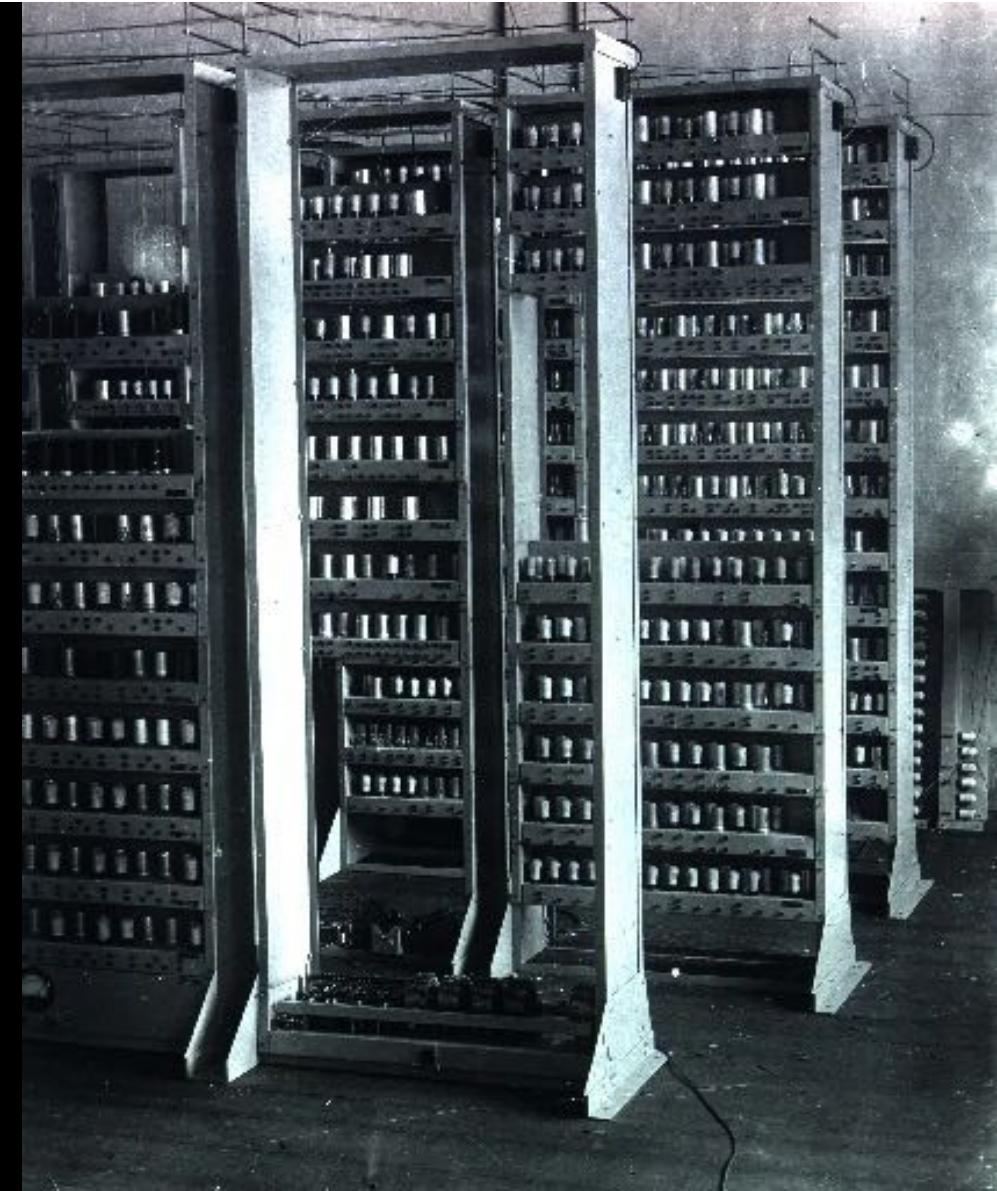
CPE10040H9

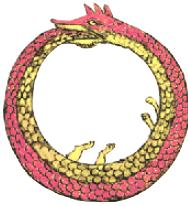


Tic Tac Toe



- Premier jeu sur ordinateur en 1952
- Electronic Delay Storage Automatic Calculator (EDSAC)



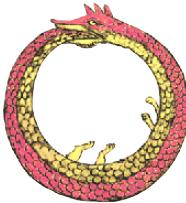


Algorithme récursif

- 2 cas triviaux arrêtent la récursion
 - 3 X ou 3 O alignés : victoire
 - 9 cases remplies : match nul
- Le cas général :
 - L'adversaire joue à son tour en utilisant le même algorithme
 - Nos objectifs sont opposés. Sa victoire est ma défaite et vice-versa.

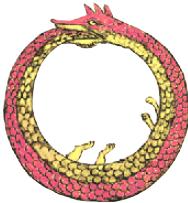


```
fonction calculeScore( case, joueur )
    marquer la case
    si la grille est gagnante pour joueur, alors
        score ← +1
    sinon si la grille est pleine, alors
        score ← 0
    sinon
        scoreAdverse ← -∞
        pour toute case vide c
            scoreAdverse ← max(scoreAdverse,
                calculeScore(c, adversaire))
        fin pour
        score ← -1 * meilleurScore
    fin si
    effacer la marque dans la case
    retourner score
```



```
fonction calculeScore( case, joueur )
    marquer la case
    si la grille est gagnante pour joueur, alors
        score ← +1
    sinon si la grille est pleine, alors
        score ← 0
    sinon
        scoreAdverse ← -∞
        pour toute case vide c
            scoreAdverse ← max(scoreAdverse,
                calculeScore(c, adversaire))
        fin pour
        score ← -1 * meilleurScore
    fin si
    effacer la marque dans la case
    retourner score
```

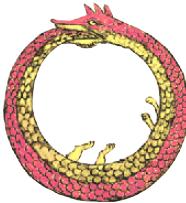
Cas triviaux



```
fonction calculeScore( case, joueur )
    marquer la case
    si la grille est gagnante pour joueur, alors
        score ← +1
    sinon si la grille est pleine, alors
        score ← 0
    sinon
        scoreAdverse ← -∞
        pour toute case vide c
            scoreAdverse ← max(scoreAdverse,
                calculeScore(c, adversaire))
        fin pour
        score ← -1 * meilleurScore
    fin si
    effacer la marque dans la case
    retourner score
```

Cas triviaux

Cas général



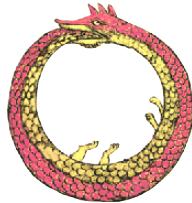
```
fonction calculeScore( case, joueur )
    marquer la case
    si la grille est gagnante pour joueur, alors
        score ← +1
    sinon si la grille est pleine, alors
        score ← 0
    sinon
        scoreAdverse ← -∞
        pour toute case vide c
            scoreAdverse ← max(scoreAdverse,
                calculeScore(c, adversaire))
        fin pour
        score ← -1 * meilleurScore
    fin si
    effacer la marque dans la case
    retourner score
```

Cas triviaux

Cas général

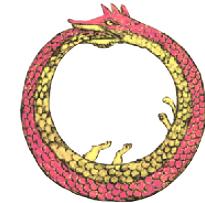
Voir algorithme permutations

Complexité de ce type d'algorithme ?



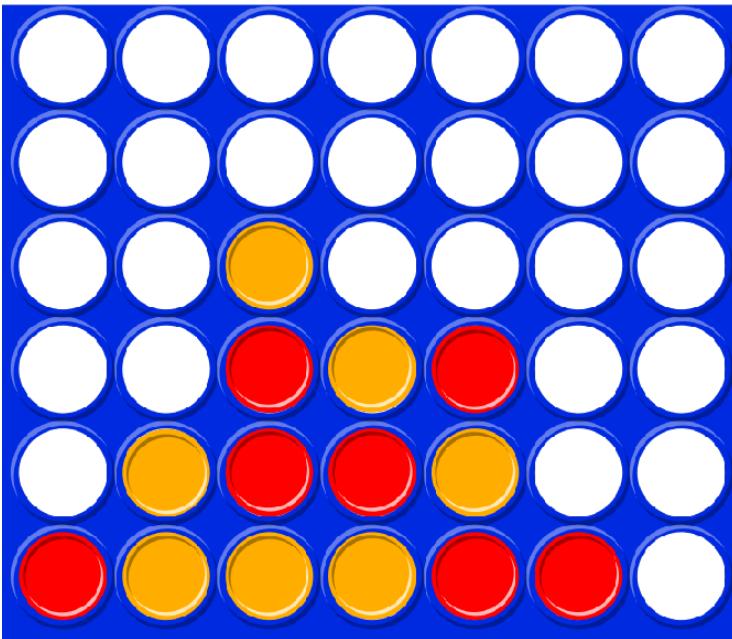
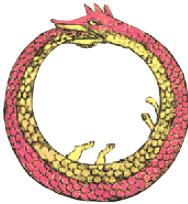
- Le joueur peut choisir entre M coups, chacune requérant un appel récursif.
- On explore N (demi-)coups successifs
- il y aura $O(M^N)$ appels récursifs.
- M et N dépendent
 - des règles du jeu
 - de la position dans le jeu.

Tic Tac Toe



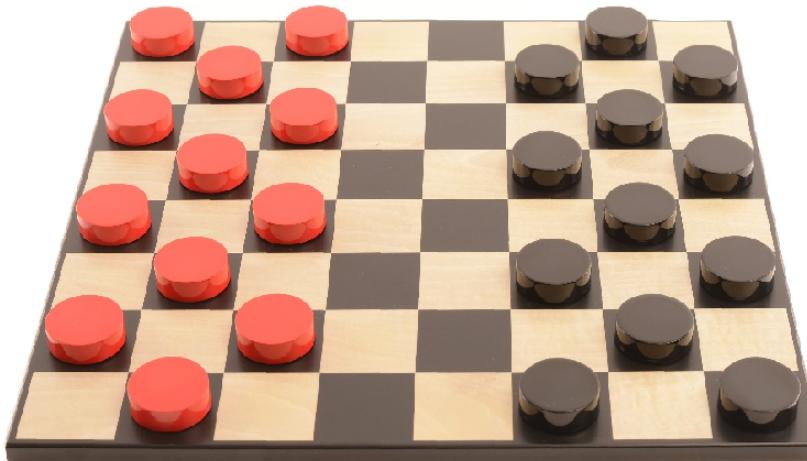
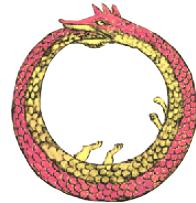
- $9! = 362880$ manières différentes de remplir la grille.
- Seulement 255168 parties possibles.
 - 131184 victoires de X
 - 77904 victoires de O
 - 46080 égalités
- Toujours égalité si les deux joueurs jouent parfaitement

Puissance 4



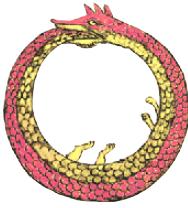
- Au début, 7 colonnes, donc $O(7^n)$ pour n coups.
- Résolu par James D. Allen en 1988.
- 4'531'985'219'092 parties possibles.
- Joueur 1 gagne au 41ème coup s'il joue parfaitement en entamant dans la colonne centrale.
- Egalité s'il entame dans un colonne entourant le centre.
- Joueur 2 gagne sinon

Jeu de dames

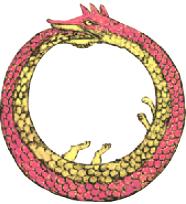


- 5×10^{20} positions possibles sur grille 8x8
- Le programme Chinook gagne le championnat du monde en 1994 face aux humains.
- Résolu par Schaeffer et al. en 2007. Environ 18 ans de calcul non-stop
- Egalité si les deux joueurs jouent parfaitement

Reversi



- Pas de solution complète ni de stratégie parfaite
- On dispose d'heuristiques suffisamment bonnes pour jouer aussi bien ou mieux que des humains
- En 1997, le logiciel Logistello gagne 6 victoires à 0 face au champion du monde humain Takeshi Murakami



Jeu d'échecs



- En 1997, Deep Blue - qui tourne sur un super ordinateur avec du hardware dédié aux échecs - bat Gary Kasparov, champion du monde humain
- En 2003, Deep Junior et X3D Fritz - qui tournent sur des PC - font tous deux match nul face à Kasparov

Jeu de Go



- Longtemps considéré comme hors d'atteinte des ordinateurs de par le nombre de combinaisons possibles: 10^{170} .
- En 2016 le logiciel AlphaGo bat le champion du monde Lee Sedol

Jeu de Go



- Longtemps considéré comme hors d'atteinte des ordinateurs de par le nombre de combinaisons possibles: 10^{170} .
- En 2016 le logiciel AlphaGo bat le champion du monde Lee Sedol

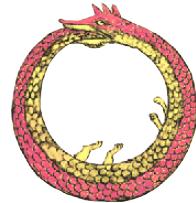


AlphaGo vs Lee Sedol

人類と人工知能の叡智をかけた
史上最強の五番勝負が始まる。
勝つのは、李世乭か？ Googleか？
賞金100万ドル

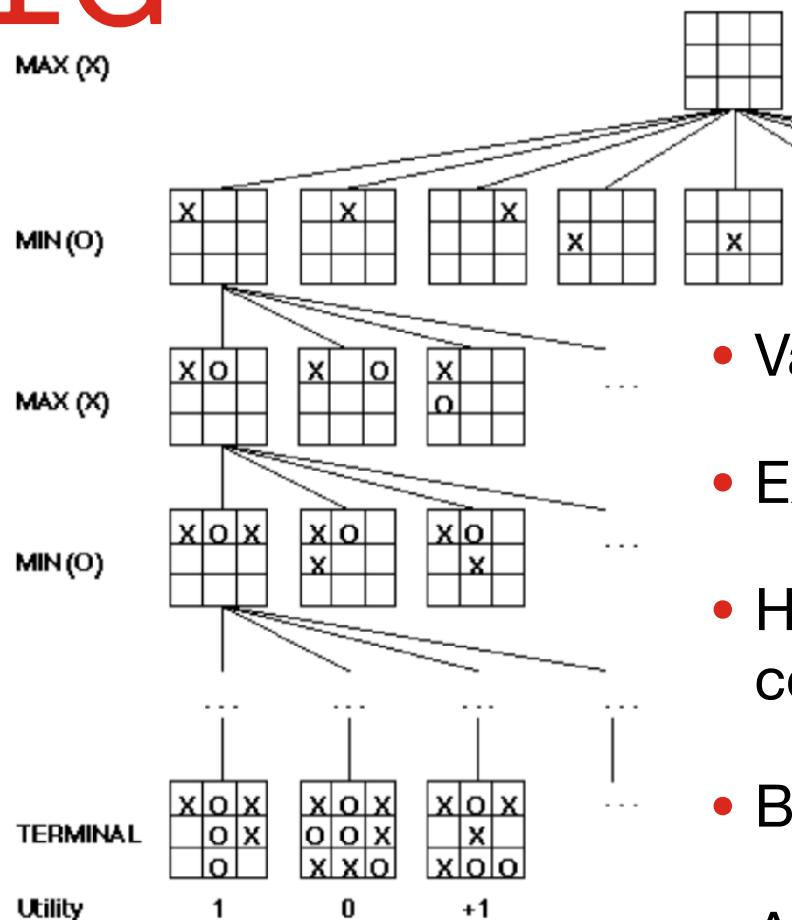
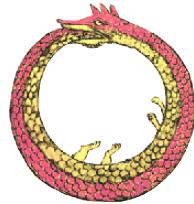
対局日程：3月9、10、12、13、15日／対局場：韓国ソウル

Comment jouent-ils si bien ?



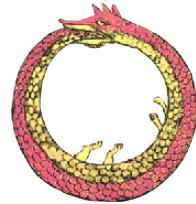
- Variantes de l'algorithme MiniMax
- Exploration limitée à une profondeur maximale
- Heuristique pour évaluer les positions non concluantes.
- Basée sur des stratégies connues
- Apprise (TD-Gammon)

Comment jouent-ils si bien ?



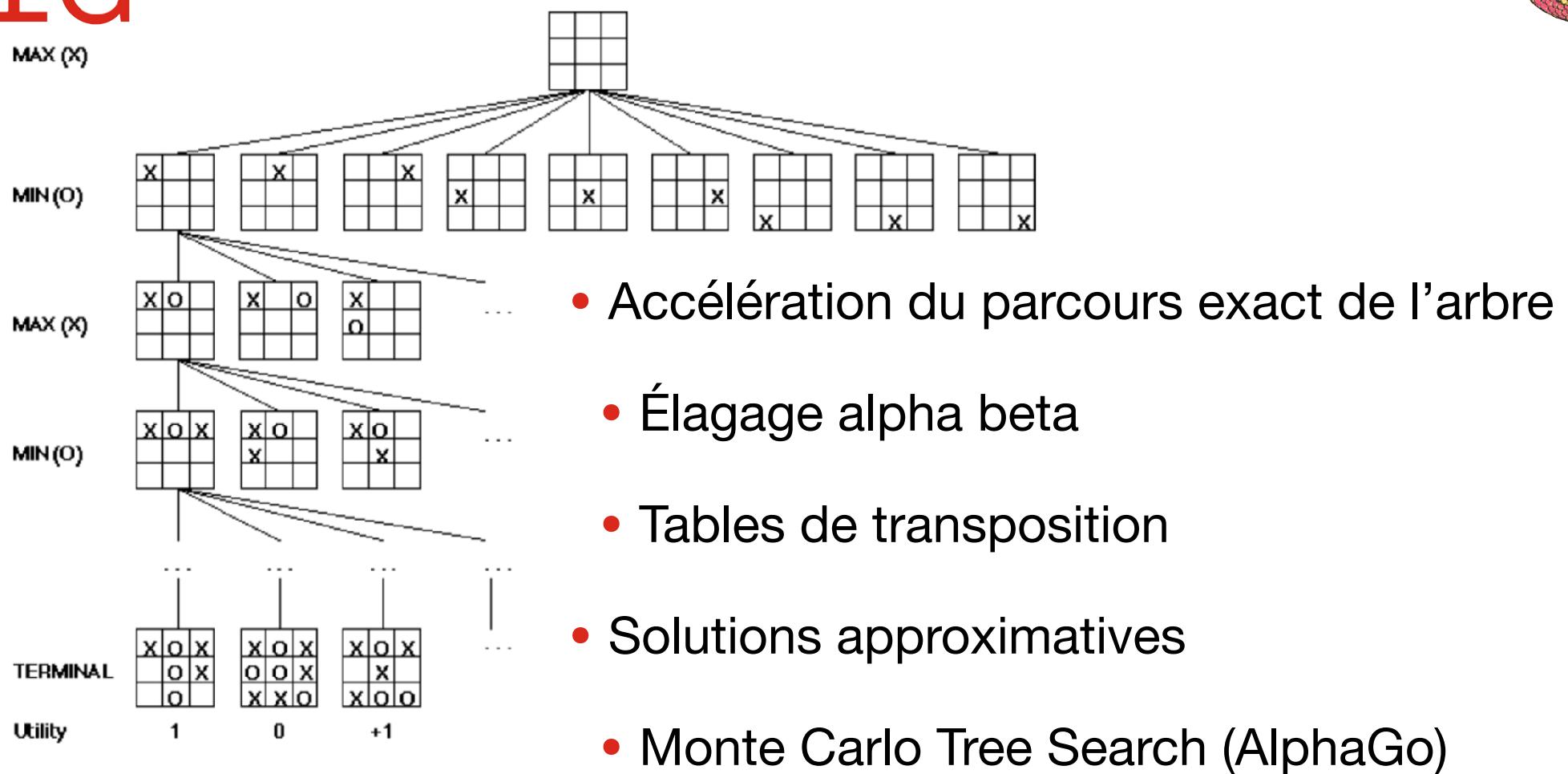
- Variantes de l'algorithme MiniMax
- Exploration limitée à une profondeur maximale
- Heuristique pour évaluer les positions non concluantes.
- Basée sur des stratégies connues
- Apprise (TD-Gammon)

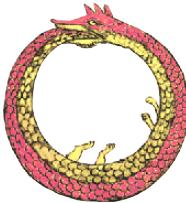
Comment jouent-ils si bien ? (2)



- Accélération du parcours exact de l'arbre
 - Élagage alpha beta
 - Tables de transposition
- Solutions approximatives
 - Monte Carlo Tree Search (AlphaGo)

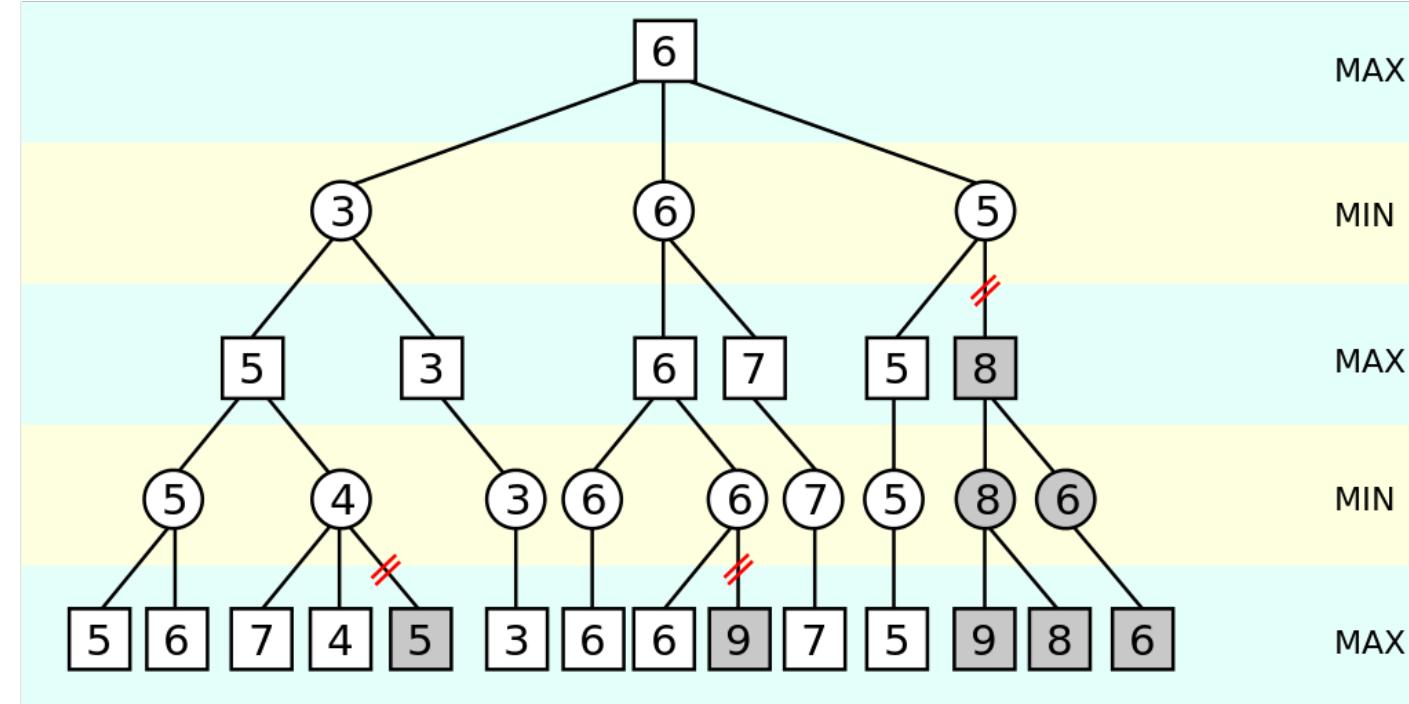
Comment jouent-ils si bien ? (2)





Elagage alpha-beta

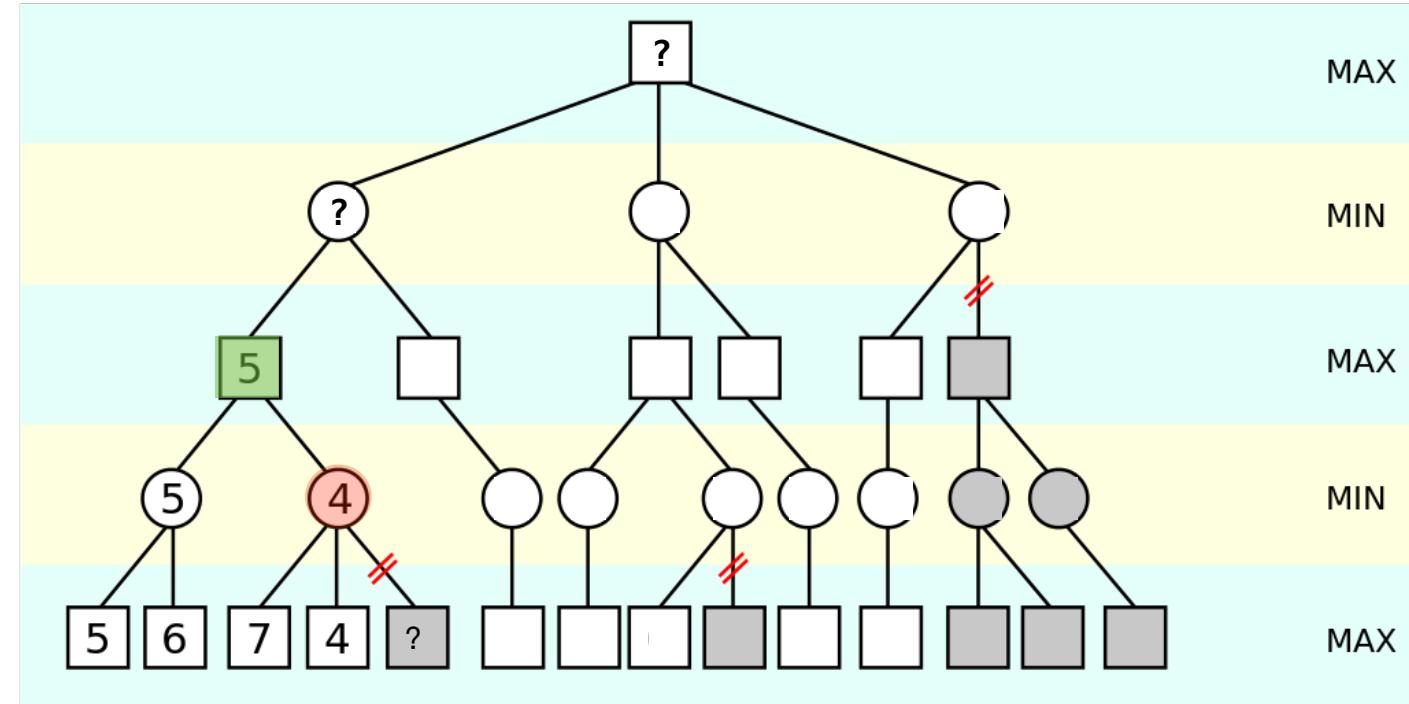
- Arrêter la recherche de mininum beta si il est déjà plus petit que alpha, le max au niveau au dessus
- Arrêter la recherche de maximum alpha si il est déjà plus grand que beta, le min au niveau au dessus



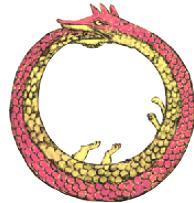
Arrêtons-nous au calcul de 4



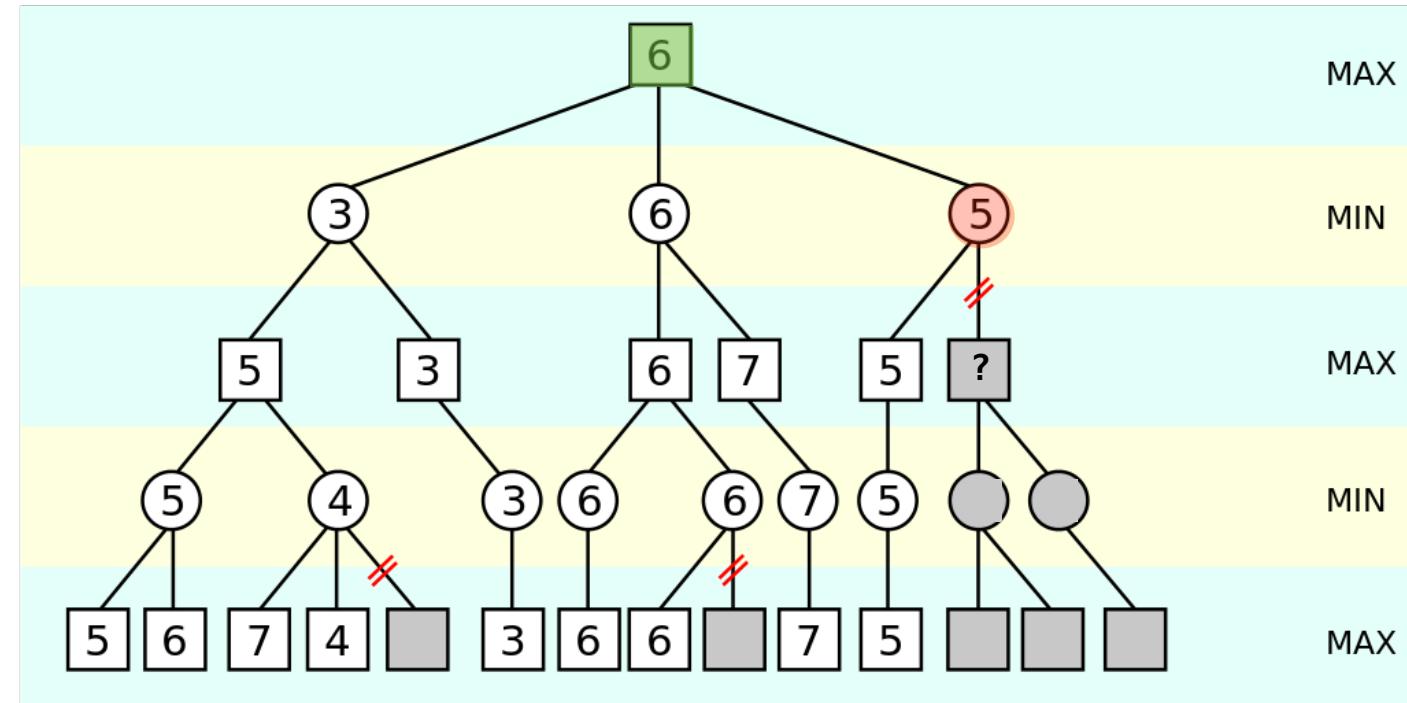
- C'est un calcul de minimum.
- Sa valeur ne peut que diminuer, avant d'évaluer ?, on sait qu'elle est ≤ 4
- La valeur de 4 ne sert qu'à mettre à jour le calcul du maximum au niveau supérieur
- Comme $4 \leq 5$, la valeur de 5 ne changera pas quelque soit la valeur de 4



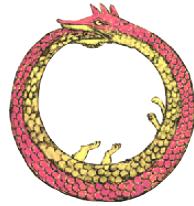
Arrêtons-nous au calcul de 5



- C'est un calcul de minimum.
 - Sa valeur ne peut que diminuer, avant d'évaluer on sait qu'elle est \leq 5 ?
 - La valeur de 5 ne sert qu'à mettre à jour le calcul du maximum au niveau supérieur
 - Comme 5 \leq 6, la valeur de 6 ne changera pas quelque soit la valeur de 5

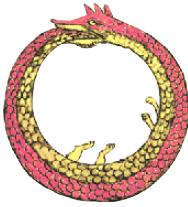


Gain lié à l'élagage ?



- Cela dépend de l'ordre dans lequel les coups sont considérés.
- Dans le meilleur cas, il permet de doubler la profondeur d'exploration
- Dans le pire cas, il n'apporte aucune amélioration
- Trouver une bonne heuristique pour l'ordre permet d'accélérer considérablement l'algorithme.

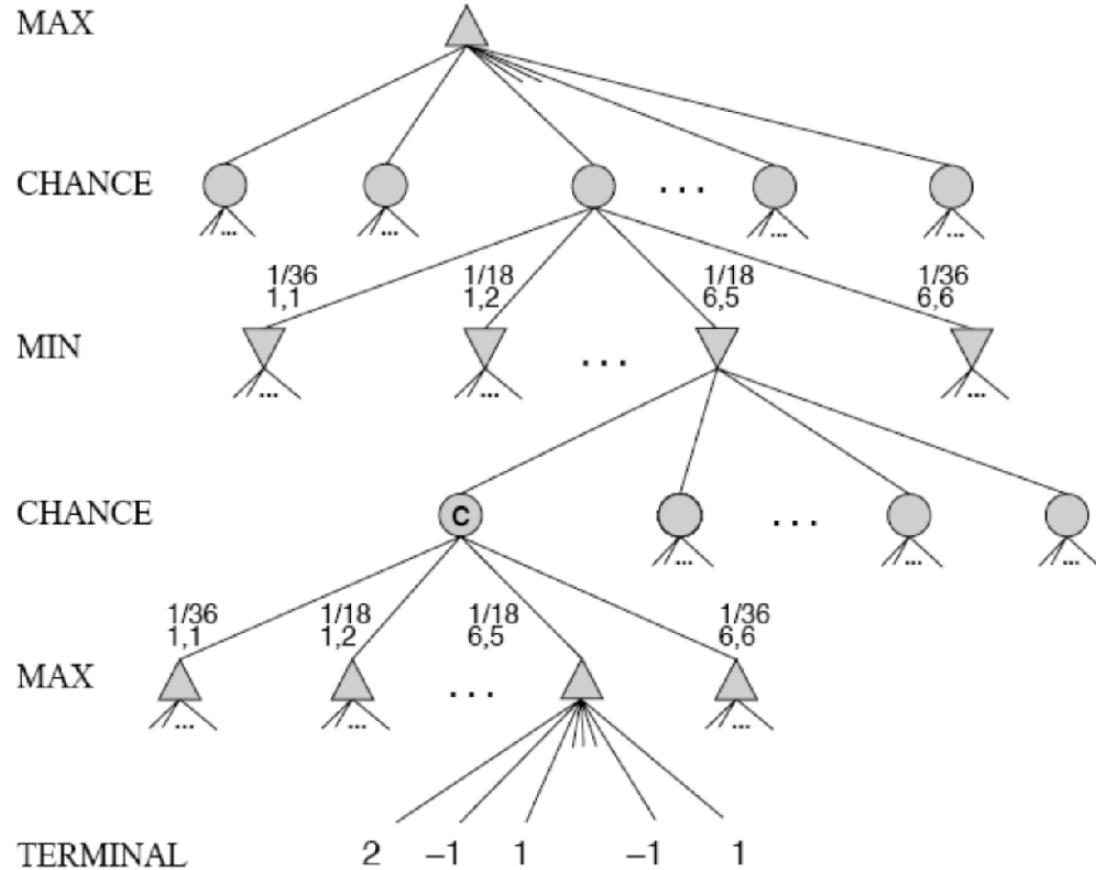
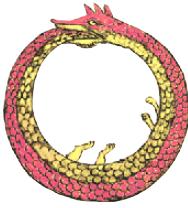
Backgammon



- Le lancer de dés introduit du hazard dans l'arbre des coups successifs possibles
- On intercale des moyennes pondérées par les probabilités $1/18$ ou $1/36$ entre les noeuds « Min » et « Max » de l'algorithme



Minimax pour backgammon

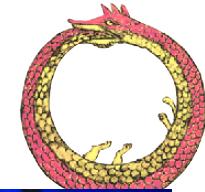


Texas Hold'em à 2 joueurs



- La version la plus simple du jeu - Texas Holdem, 2 joueurs, mises limitées - est résolue en 2015
- Le programme Cepheus y joue parfaitement. Il utilise 12 terabytes de données précalculées

Texas Hold'Em à 6 joueurs

Brown *et al.*, *Science* **365**, 885–890 (2019)

30 August 2019

<https://doi.org/10.1126/science.aay2400>

RESEARCH

RESEARCH ARTICLE

COMPUTER SCIENCE

Superhuman AI for multiplayer poker

Noam Brown^{1,2*} and Tuomas Sandholm^{1,3,4,5*}

In recent years there have been great strides in artificial intelligence (AI), with games often serving as challenge problems, benchmarks, and milestones for progress. Poker has served for decades as such a challenge problem. Past successes in such benchmarks, including poker, have been limited to two-player games. However, poker in particular is traditionally played with more than two players. Multiplayer games present fundamental additional issues beyond those in two-player games, and multiplayer poker is a recognized AI milestone. In this paper we present *Pluribus*, an AI that we show is stronger than top human professionals in six-player no-limit Texas hold'em poker, the most popular form of poker played by humans.

Poker has served as a challenge problem for the fields of artificial intelligence (AI) and game theory for decades (1). In fact, the foundational papers on game theory used poker to illustrate their concepts (2, 3). The reason for this choice is simple: No other popular recreational game captures the challenges of hidden information as effectively and as elegantly as poker. Although poker has been useful as a benchmark for new AI and game-theoretic techniques, the challenge of hidden information in strategic settings is not limited to recreational games. The equilibrium

by, for example, trying to detect and exploit weaknesses in the opponent. A Nash equilibrium is a list of strategies, one for each player, in which no player can improve by deviating to a different strategy. Nash equilibria have been proven to exist in all finite games—and many infinite games—though finding an equilibrium may be difficult.

Two-player zero-sum games are a special class of games in which Nash equilibria also have an extremely useful additional property: Any player who chooses to use a Nash equilibrium is guaranteed to not lose in expectation no matter what

exact Nash equilibrium. For example, the Nash equilibrium strategy for Rock-Paper-Scissors is to randomly pick Rock, Paper, or Scissors with equal probability. Against such a strategy, the best that an opponent can do in expectation is tie (10). In this simple case, playing the Nash equilibrium also guarantees that the player will not win in expectation. However, in more complex games, even determining how to tie against a Nash equilibrium may be difficult; if the opponent ever chooses suboptimal actions, then playing the Nash equilibrium will indeed result in victory in expectation.

In principle, playing the Nash equilibrium can be combined with opponent exploitation by initially playing the equilibrium strategy and then over time shifting to a strategy that exploits the opponent's observed weaknesses (for example, by switching to always playing Paper against an opponent that always plays Rock) (11). However, except in certain restricted ways (12), shifting to an exploitative nonequilibrium strategy opens oneself up to exploitation because the opponent could also change strategies at any moment. Additionally, existing techniques for opponent exploitation require too many samples to be competitive with human ability outside of small games. *Pluribus* plays a fixed strategy that does not adapt to the observed tendencies of the opponents.

Although a Nash equilibrium strategy is guaranteed to exist in any finite game, efficient algorithms for finding one are only proven to exist for special classes of games, among which two-player zero-sum games are the most prom-



Chris Ferguson



Darren Elias

2.6.

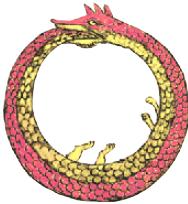
Récursif

?

Itératif



Factorielle

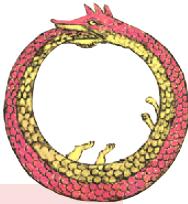


- Avec un seul appel récursif, la fonction récursive est équivalente à une simple boucle.
- Attention, les calculs sont effectué dans l'ordre inverse
 - Récursif : (((((1).2).3)....(n-1)).n)
 - Itératif : 1.n.(n-1)....3.2

```
fonction factorielle(n)
    si n vaut 0 alors
        retourner 1
    sinon
        retourner
            n x factorielle(n-1)
    fin si
```

```
fonction factorielle(n)
    r <- 1
    tant que n > 1
        r <- r x n
        décrémenter n
    fin tant que
    retourner r
```

Fibonacci

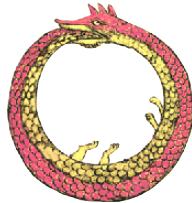


- Version itérative de Fibonacci plus complexe à écrire
- Complexité itérative bien meilleure
 - Récursif en $O(\phi^n)$
 - Itératif en $O(n)$

```
fonction Fibonacci(n)
    si n vaut 0 ou 1 alors
        retourner n
    sinon
        retourner Fibonacci(n-1)
            + Fibonacci(n-2)
    fin si
```

```
fonction Fibonacci(n)
    si n vaut 0 ou 1 alors
        retourner n
    Fn-1 ← 0
    Fn ← 1
    pour i de 2 à n
        Fn-2 ← Fn-1
        Fn-1 ← Fn
        Fn ← Fn-1 + Fn-2
    fin pour
    retourner Fn
```

Somme des n premiers entiers



- L'algorithme du chapitre 1 peut-être écrit itérativement, récursivement ou en utilisant la formule de Gauss
- L'approche récursive est-elle une bonne idée ?

```
long sumRec(int n) {  
    if(n<=0) return 0;  
    return n + sumRec(n-1);  
}
```

```
long sumIter(int n) {  
    long sum = 0;  
    for (int i=0; i<=n; ++i)  
        sum += i;  
    return sum;  
}
```

```
long sumGauss(int n) {  
    return (long)n*(n+1)/2;  
}
```

The screenshot shows the CLion IDE interface with the project "playground" open. The main editor window displays the file `main.cpp` containing the following C++ code:

```
5     using namespace std;
6
7     long sumRec(int n) {    n: 25267
8         if(n<=0) return 0;
9         return n + sumRec(n-1);    n: 25267
10    }
11
12 int main() {
13     cout << sumRec( n: 200000);
14 }
```

The line `return n + sumRec(n-1);` at line 9 is highlighted with a red arrow and a lightning bolt icon, indicating it is the current line of execution. The variable `n` is highlighted in blue with a value of `25267`.

In the bottom-left corner of the editor, there is a small floating window with the text `f sumRec`.

The bottom part of the interface is the Debug tool window, which includes tabs for Debugger, Console, Variables, LLDB, Memory View, and Watches.

- Debugger Tab:** Shows the current thread as `Thread 1` and the stack frames for `sumRec(int)` at line 9.
- Variables Tab:** Displays the exception information:
 - Exception:** `EXC_BAD_ACCESS (code=2, address=0x7ffee048dff8)`
 - Locals:** `n = {int} 25267`
- Watches Tab:** Shows the message `No watches`.

The status bar at the bottom of the screen shows the text `100% CPU usage`.

Pile d'appels (call stack)



- A chaque appel de fonction le programme empile dans une structure de pile
 - Les variables locales de fonction
 - L'adresse de retour où sauter en sortant de la fonction
- A chaque sortie de fonction, le programme dépile les informations liées à cette fonction et saute à l'adresse indiquée
- Une récursion trop profonde peut dépasser la capacité de cette pile.
- Et pourtant ...

The screenshot shows a C++ development environment with the following details:

- Project Bar:** Shows the project name "SommeRecursive" and build configuration "Release".
- Toolbars:** Standard file operations (New, Open, Save, Print, Find, Copy, Paste) and other developer tools.
- Left Sidebar:** Includes tabs for "Project", "Structure", and "Favorites".
- Code Editor:** Displays the file "main.cpp" containing the following code:

```
1 #include <iostream>
2 #include <vector>
3 #include <numeric>
4
5 using namespace std;
6
7 long sumRec(int n) {
8     if(n<=0) return 0;
9     return n + sumRec( n: n-1);
10 }
11
12 int main() {
13     cout << sumRec( n: 200000);
14 }
```
- Run Tab:** Shows the run configuration "SommeRecursive" and the output window.
 - Output:** Displays the command run and the resulting output:

```
/Users/oce/OneDrive/GitHub/ocuisenaire/playground/cmake-build-release/SommeRecursive
2000010000
Process finished with exit code 0
```
 - Toolbar:** Includes icons for Run, Stop, Build, and Favorites.
- Bottom Navigation:** Standard navigation tabs: Run, Debug, Problems, Terminal, CMake, Messages, TODO, Event Log, and a status bar indicating "A" (Active).

The screenshot shows a C++ development environment with the following details:

- Project:** playground
- File:** main.cpp
- Code Content:**

```
1 #include <iostream>
2 #include <vector>
3 #include <numeric>
4
5 using namespace std;
6
7 long sumRec(int n) {
8     if(n<=0) return 0;
9     return n + sumRec( n: n-1);
10 }
11
12 int main() {
13     cout << sumRec( n: 200000);
14 }
```
- Run Output:**

```
sumRec
Run: SommeRecursive
/Users/oce/OneDrive/GitHub/ocuisenaire/playground/cmake-build-release/SommeRecursive
2000010000
Process finished with exit code 0
```

A large watermark "live | Release" with a downward arrow is overlaid on the right side of the interface.

Récursion finale (tail récursion)

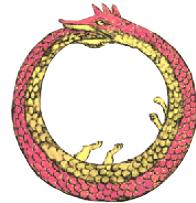


- Quand la toute dernière opération de la fonction récursive est l'appel récursif
- Pas besoin d'empiler de nouvelles infos
- On peut réutiliser celles de la fonction courante
- Mais ... pas en mode debug

```
long sumRec(int n) {  
    if(n<=0) return 0;  
    return n + sumRec(n-1);  
}
```

```
long sumTailRec(int n, long r) {  
    if(n==0) return r;  
    return sumTailRec(n-1,n+r);  
}
```

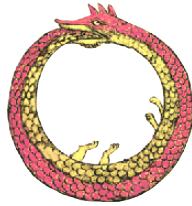
Itération vs. récursion terminale



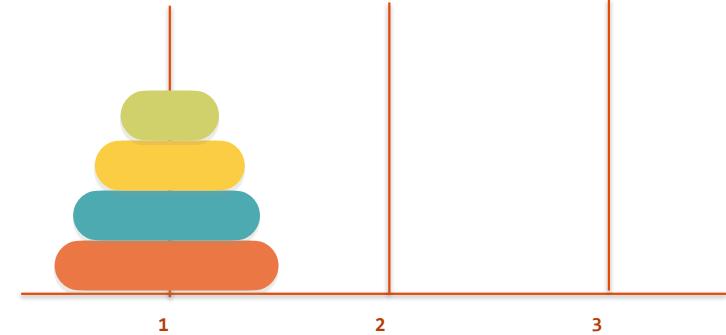
```
fonction recursion(n)
    faire A
    si B, alors
        faire C
        appeler recursion(..)
    fin si
```

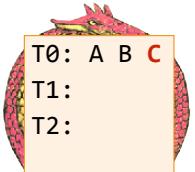
```
fonction iteration(n)
    boucler
        faire A
        si non B, alors
            sortir boucle
        fin si
        faire C
    fin boucle
```

Tour de Hanoï ?



- En général, les fonctions avec plusieurs appels récursifs sont difficiles / impossibles à rendre itératives
- Pour les tours de Hanoï, on dispose cependant d'un algorithme alternatif
 - Le + petit disque bouge une fois sur 2, toujours dans le même sens
 - L'autre déplacement est imposé (le + petit disque va sur le + grand)





Tour de Hanoï itératif

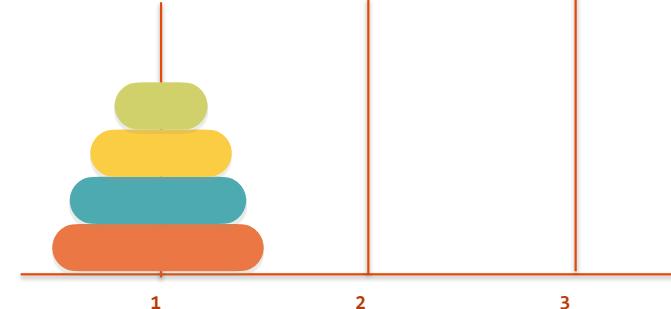
fonction Hanoi(n)

bouger le plus petit disque
vers la droite (n pair)
ou la gauche (n impair)

boucler $2^{n-1}-1$ fois

bouger le disque qui n'est pas le plus petit
bouger le plus petit disque
vers la droite (n pair)
ou la gauche (n impair)

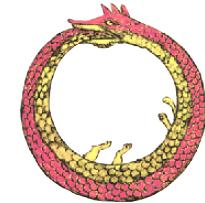
fin boucler



T0: A B C
T1:
T2:
T0: A B
T1:
T2: C
T0: A
T1: B
T2: C
T0: A
T1: B C
T2:
T0: A
T1: B C
T2: A
T0: C
T1: B
T2: A
T0: C
T1:
T2: A B
T0:
T1:
T2: A B C

Algorithmes

Complexités



Factorielle récursif	$O(n)$
Factorielle itératif	$O(n)$
Fibonacci récursif	$O(1.618^n)$
Fibonacci itératif	$O(n)$
PGCD (Euclide)	$O(\log(n))$
Tours de Hanoï récursif	$O(2^n)$
Tours de Hanoï itératif	$O(2^n)$
Permutations	$O(n!)$
Tic Tac Toe	$9!$
Puissance 4, profondeur d'exploration de d tours	$O(7^d)$
Minimax (negamax), m mouvements possibles par tour, profondeur de d tours	$O(m^d)$