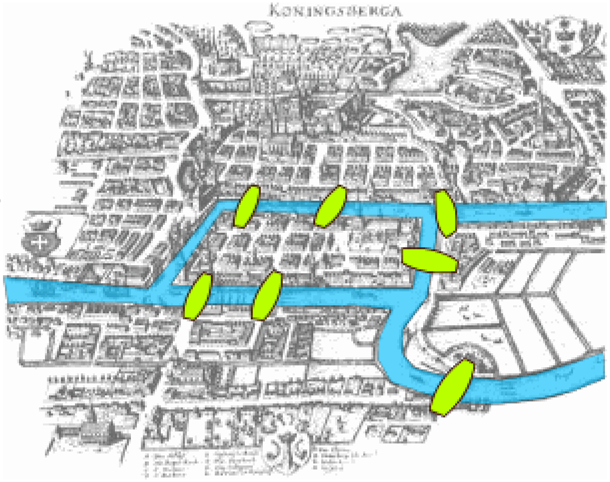
Graph Theory

- Voir [Wes22; Chapter 1-2]
- ou [CLRS22; Chapter 22-24]

[Wes22] D. B. West. *Introduction to Graph Theory* (2022).
[CLRS22] T. H. Cormen, C. E. Leiserson, R. L. Rivest and C. Stein. *Introduction to Algorithms, Fourth Edition* (MIT Press, 2022).

Seven Bridges of Königsberg

Est-il possible de prendre tous les ponts de la ville une et une seule fois (le point de départ est au choix) ?

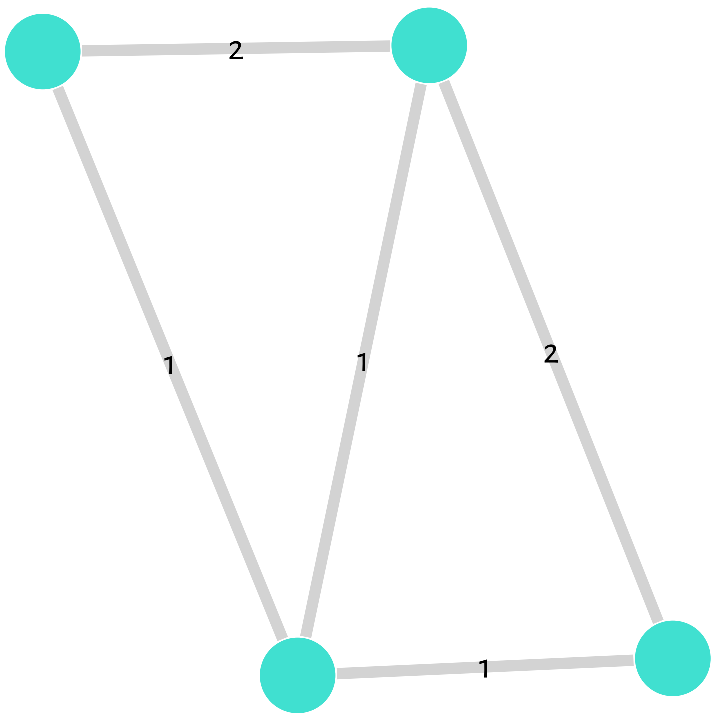


Définition

Un graph a un ensemble V de noeuds (nodes) / sommets (vertices) et E d'arêtes (edges) / arcs.

Math	Informatique	Non-dirigé	Dirigé
V	`Nodes`	`Sommet / vertex`	`Noeud / node`
E	`Edges`	`Arête / edge`	`Arc`

Le graphe du problème de Königsberg test:



Graphes et polyhèdres

Prenons un polyhèdre, où se cache le graphe ?

On associe un sommet du graphe à chaque sommet du polyhèdre et une arêtes à chaque arêtes (c'est la même terminologie, coincidence ?).

Relation d'Euler: le nombre de faces est égal à $2 - |V| + |E|$.

Terminologie des parcours

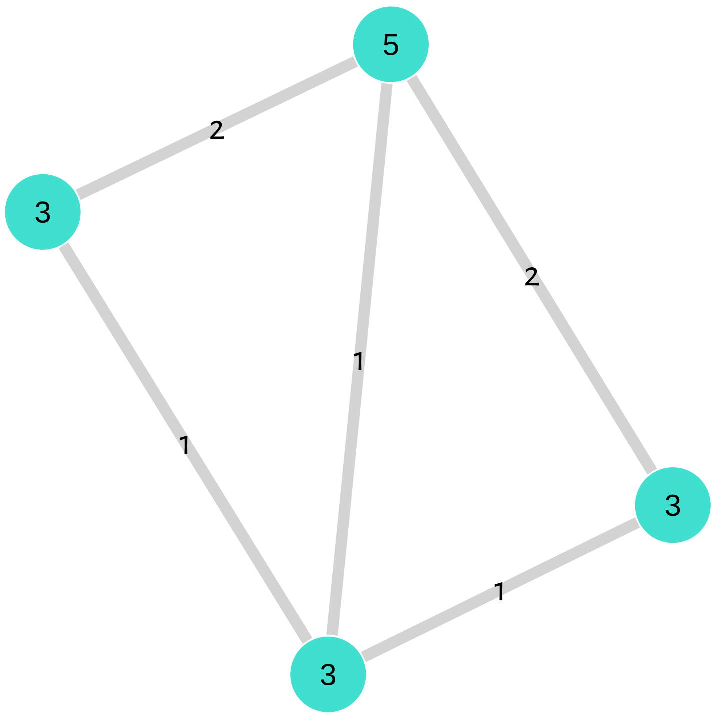
Un parcours / walk de longueur k d'un graphe $G = (V, E)$ est une suite de $k + 1$ noeuds $v_0, v_1, \ldots, v_k \in V$ et k arêtes $(v_0, v_1), (v_1, v_2), \ldots, (v_{k-1}, v_k) \in E$. Si $v_0 = v_k$, le parcours est fermé.

	Noeuds distincts	**Arêtes distinctes**
`Ouvert`	`Chemin`	`Piste / Trail`
`Fermé`	`Cycle`	`Circuit`

Piste (resp. circuit) *Eulérienne* : Une piste (resp. circuit) qui visite **toutes** les arêtes.

Degré d'un noeud

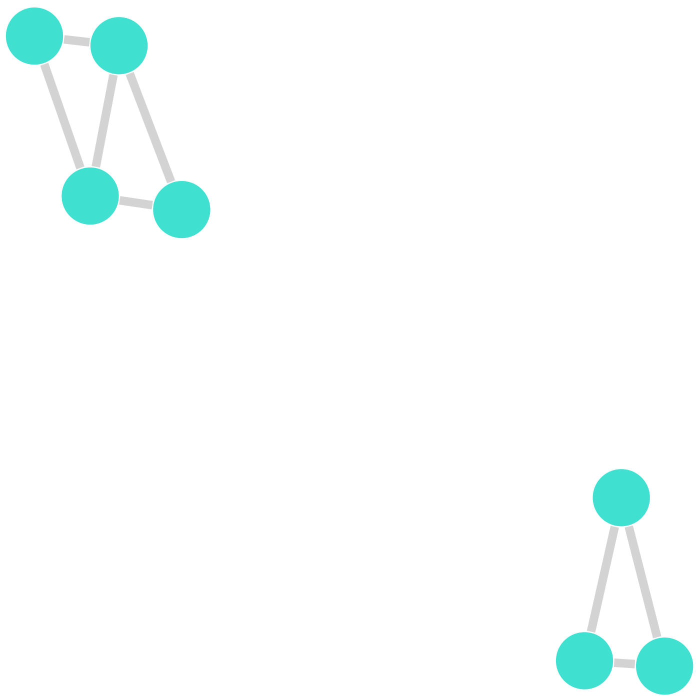
Le degré d'un noeud v est le nombre d'arête qui lui sont incidentes.




```
let
    g = Graph(4)
    add_edge!(g, 1, 2)
    add_edge!(g, 1, 4)
    add_edge!(g, 2, 3)
    add_edge!(g, 2, 4)
    add_edge!(g, 3, 4)
    gplot(g, nodelabel = [3, 3, 3, 5], edgelabel = [1, 2, 1, 1, 2])
end
```

Composante connexe

Une composante connexe C est un ensemble de noeuds tels que toutes paires de noeuds est connectée par un chemin.



Calcul de composantes connexes

On démarre en assignant une composant connexe différente pour chaque noeud. Pour chaque arête, on fusionne les composantes connexes des deux noeuds liés par l'arête. Comment calculer cette fusion efficacement ?

Supposons qu'on ait 4 noeuds. On démarre avec le tableau $(1, 2, 3, 4)$ signifiant que chaque noeud est dans une composante connexe différente.

- En commençant avec l'arête $(1, 2)$, on fusionne et on arrive au tableau $(1, 1, 3, 4)$.
- Si on voit ensuite l'arête $(3, 4)$, on arrive alors au tableau $(1, 1, 3, 3)$.
- Si on voit ensuite l'arête $(1, 3)$, on update le tableau à $(1, 1, 1, 1)$.

▶ **Quelle est la complexité de cet algorithm ?**

Disjoint-Set datastructure

Pour améliorer l'algorithme, pour la fusion de l'algorithm $(1, 3)$, on peut updater le tableau à $(1, 1, 1, 3)$. Pour le noeud 4, on encode donc qu'il est dans la même connected component que le noeud 3 qui qui est dans la même connected component que le noeud 1. Le noeud 4 est donc lié indirectement au noeud 1 qui est appelé sa racine.

▶ **Quelle est complexité après cette amélioration ?**

De façon surprenante, si on met à jour le valeur dans le tableau pour mettre directement le root après l'avoir calculé, la complexité passe à $O(|V|\alpha(|V|))$ où α est la réciproque de la fonction d'Ackermann, une fonction qui augmente extrêmement lentement donc en pratique, c'est presque $O(|V|)$.

Piste Eulérienne

Théorème [Wes22; Theorem 1.2.26] Considérons un graphe avec une seule composante connectée. Il existe un circuit Eulérienne si et seulement si tous les noeuds ont un degré pair. Si tous les noeuds ont un degré pair sauf 2 alors il n'existe pas de circuit mais une piste Eulérienne.

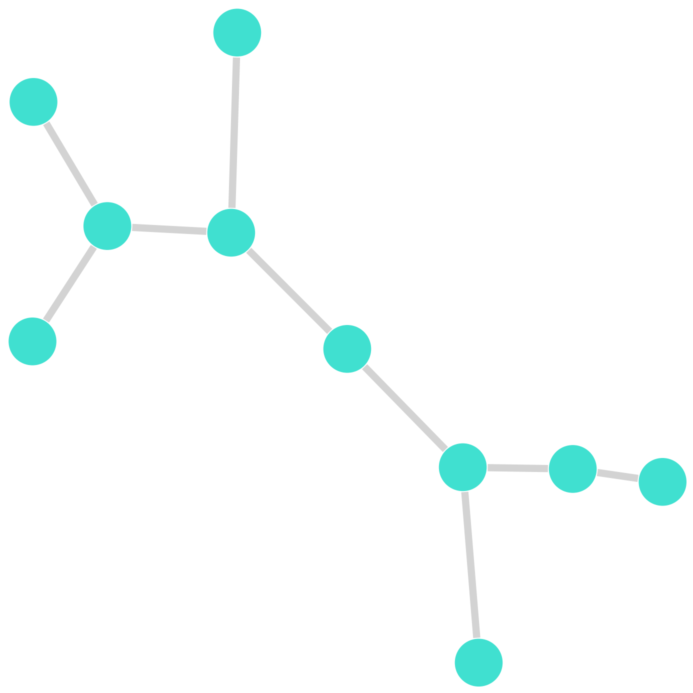
▶ **Proof**

Arbres et forêts

Définition

- Une *forêt* (forest) est un graphe qui n'a **pas** de **cycle**.
- Un *arbre* (tree) est une forêt avec **une** seule **composante** connexe.

Par abus de language, on parle souvent d'arbre sans se soucier de si le graphe est connecté ou pas.



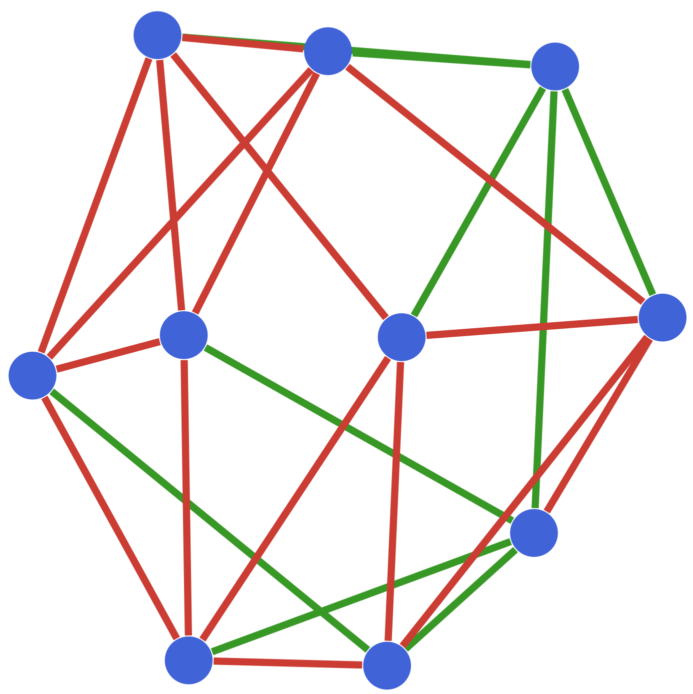
```
gplot(uniform_tree(10))
```

Spanning tree

Étant donné un graphe $G(V,E)$, sa *forêt sous-tendante* (spanning forest) est une forêt $G'(V,E')$ où $E' \subseteq E$. Comment la trouver ?

```
spanning_forest (generic function with 2 methods)
```

```
function spanning_forest(g, edges_it = edges(g))
    component = IntDisjointSets(nv(g))
    function create_loop!(edge)
        loop = in_same_set(component, src(edge), dst(edge))
        union!(component, src(edge), dst(edge))
        return loop
    end
    return [edge for edge in edges_it if !create_loop!(edge)]
end
```

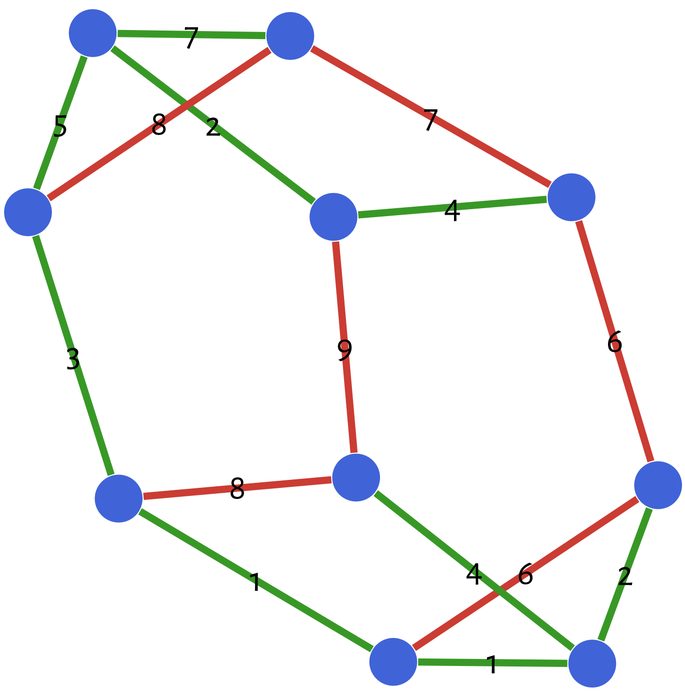


```
let
    g = random_regular_graph(10, 5)
    cols = [colors.red, colors.green]
    tree = Set(spanning_forest(g))
    gplot(g, edgestrokec = [cols[(edge in tree) + 1] for edge in edges(g)],
    nodefillc = colors.blue)
end
```

Minimum spanning tree

kruskal (generic function with 1 method)

```
kruskal(g) = spanning_forest(g, sort(collect(edges(g)), by = weight))
```

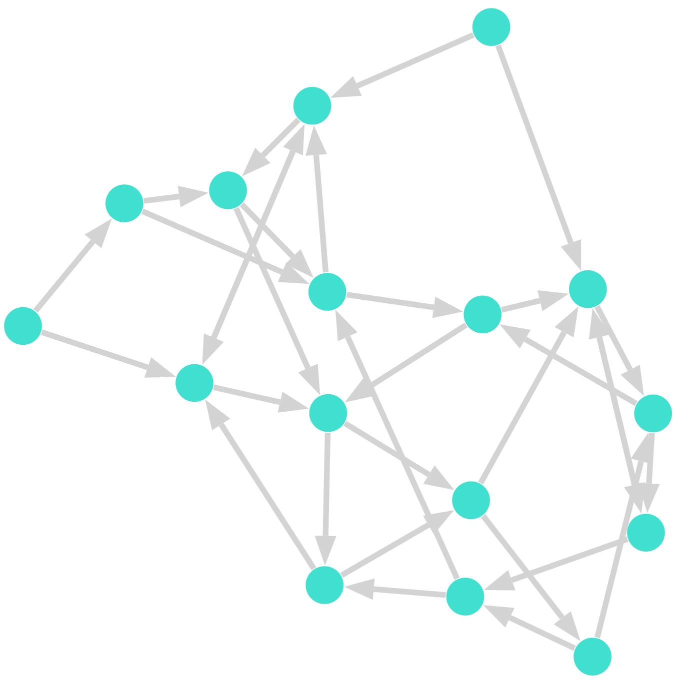


▶ **Quelle est la complexité de Kruskal ?**

Graphes dirigés

Dans un graphe dirigé, une arête (i, j) a un **sens**. Intuitivement, on peut aller de i vers j mais on ne peut qu'aller de j vers i si il y a aussi une autre arête de (j, i).

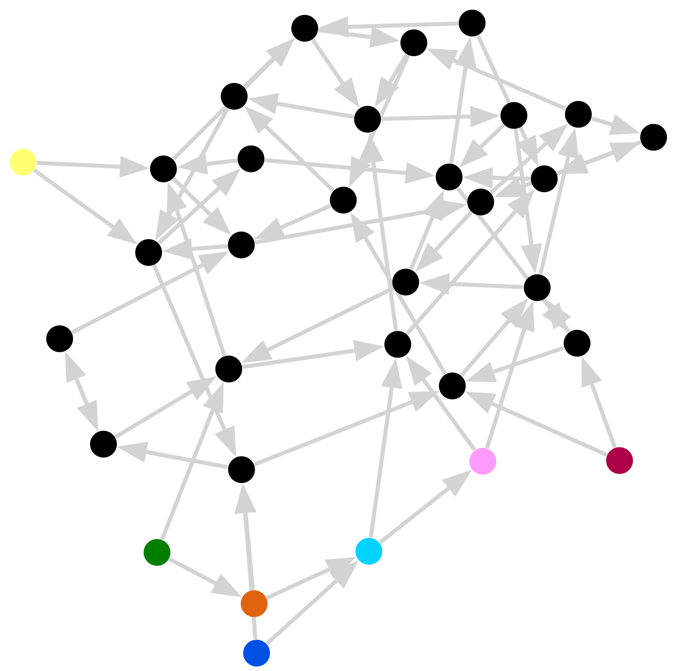
Que deviennent les notions de **connected components**, **spanning tree**, etc...



```
gplot(random_regular_digraph(16, 2))
```

Strongly connected components

Définition Une **composante fortement connexe** d'un graphe dirigé $G(V, E)$ est un ensemble $V' \subseteq V$ tel qu'il existe un chemin de u vers v pour tous $u, v \in V'$.



```
let
    Random.seed!(32)
    g = random_regular_digraph(32, 2)
    c = Set.(strongly_connected_components(g))
    cols = distinguishable_colors(length(c))
    tree = Set(spanning_forest(g))
    gplot(g, nodefillc = [cols[findfirst(comp -> v in comp, c)] for v in
    vertices(g)])
end
```

Directed Acyclic Graph (DAG)

Si un graph contient un cycle, tous les noeuds de ce cycle sont contenus dans la même composante fortement connexe. Si chaque composante fortement connexe est fusionnée en un noeud, il ne reste plus de cycle. Le graphe résultat n'a donc plus de cycle, il est dit *acyclique* (DAG).

On va voir que beaucoup de problème peuvent se voir comme un problème de calcul de chaque noeud d'un graphe où les arêtes représentent les dépendences de calcul.

Suite de Fibonacci

$$F_n = F_{n-1} + F_{n-2} \qquad F_2 = F_1 = 1$$

```
fib (generic function with 1 method)
```

```julia
function fib(n)
    if n <= 2
        return 1
    else
        return fib(n - 1) + fib(n - 2)
    end
end
```

n = 10

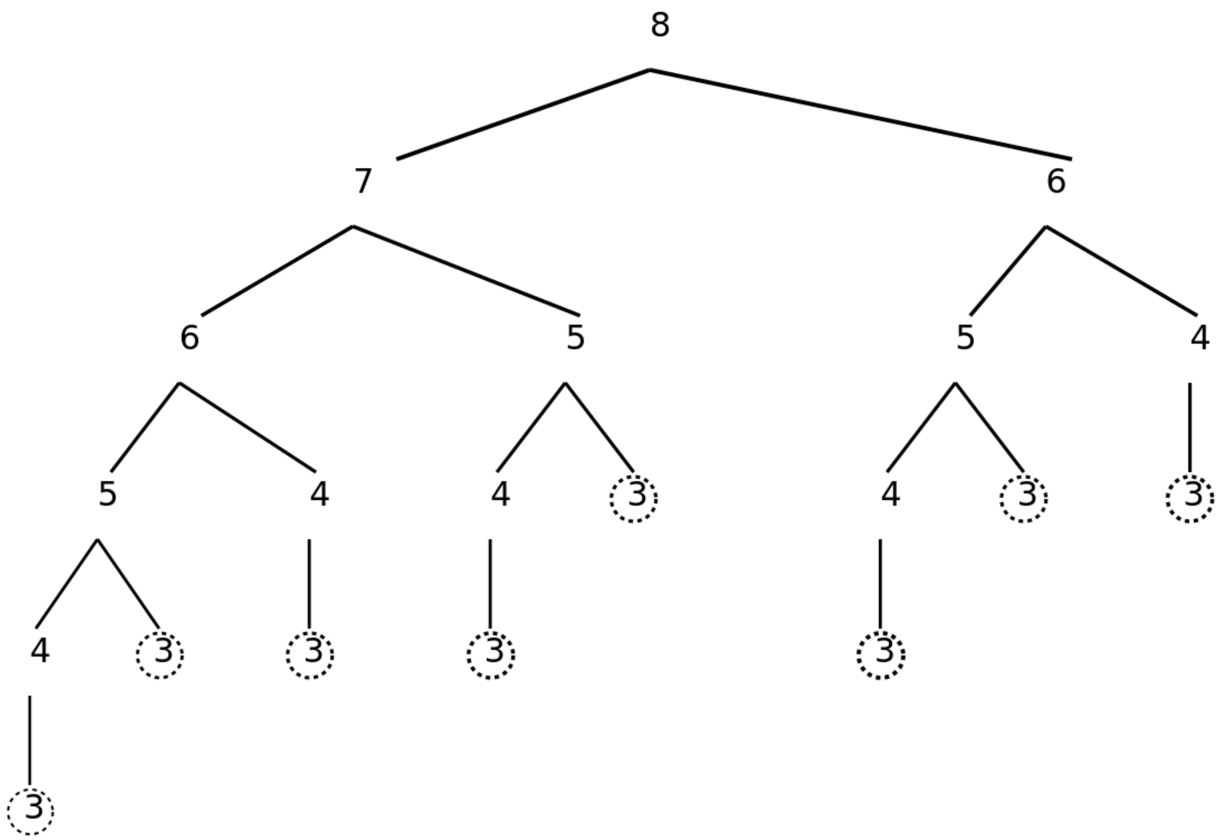
```
55
```

```julia
@time fib(n)
```

```
0.000001 seconds
```

Combien de **fois** `fib(3)` est appelé quand l'utilisateur appelle `fib(8)` ?

Visualisation des appels récursifs

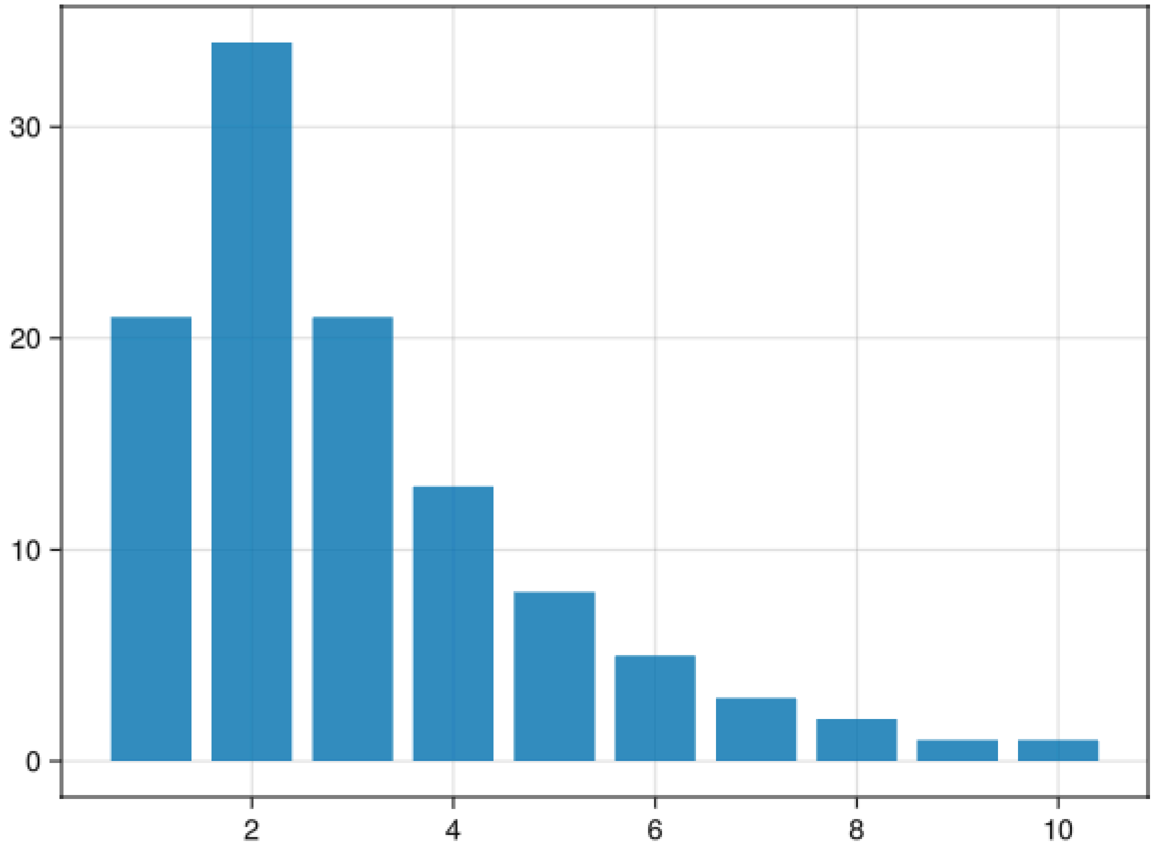



```
draw_fibo(8)
```

Nombre d'appels

n = 10

Nombre d'appels de `fib(k)` venant de `fib(n)`




```
Makie.barplot(1:n, called)
```

Bottom-Up / Dynamic Programming

Définition *Ordre topologique*: Ordre des noeuds dans lequel chaque noeud u apparait après tous les noeuds v tels qu'il y a un chemin de u vers v.

n = 10

```
fib_bottom_up (generic function with 1 method)
```

```julia
function fib_bottom_up(n)
    fib = zeros(Int, n)
    fib[1] = fib[2] = 1
    for k in 3:n
        fib[k] = fib[k - 1] + fib[k - 2]
    end
    return fib[n]
end
```

```
55
```

```julia
@time fib_bottom_up(n)
```

```
0.000002 seconds (2 allocations: 144 bytes)
```

Top-Down / Mémoïzation

n = 10

fib! (generic function with 1 method)

```
function fib!(cache, n)
    if cache[n] == 0
        cache[n] = fib!(cache, n - 1) + fib!(cache, n - 2)
    end
    return cache[n]
end
```

cached_fib (generic function with 1 method)

```
function cached_fib(n)
    cache = zeros(Int, n)
    cache[1] = cache[2] = 1
    return fib!(cache, n)
end
```

55

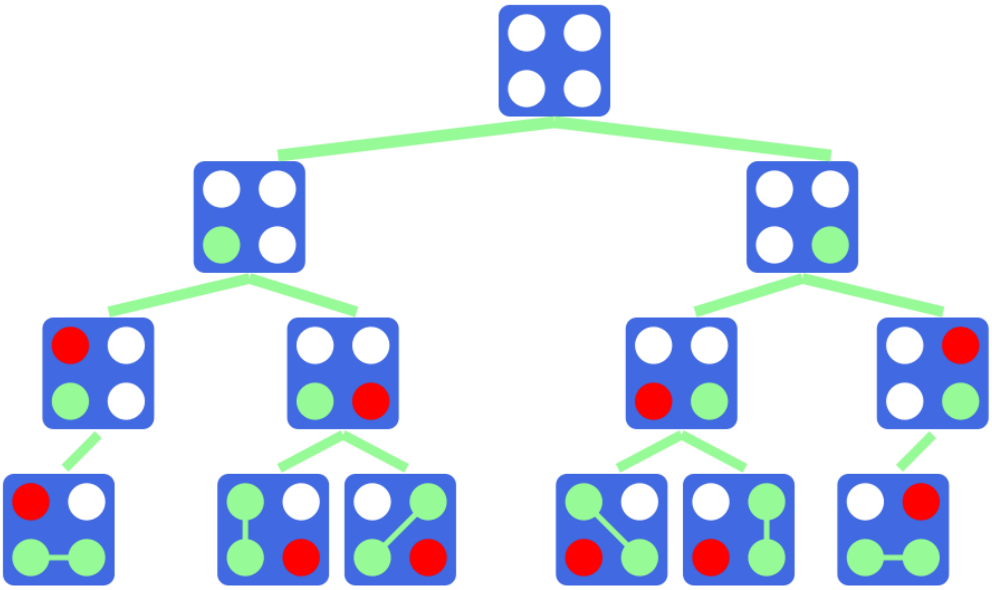
```
@time cached_fib(n)
```

```
0.000003 seconds (2 allocations: 144 bytes)
```

Une IA pour le jeu Puissance 4

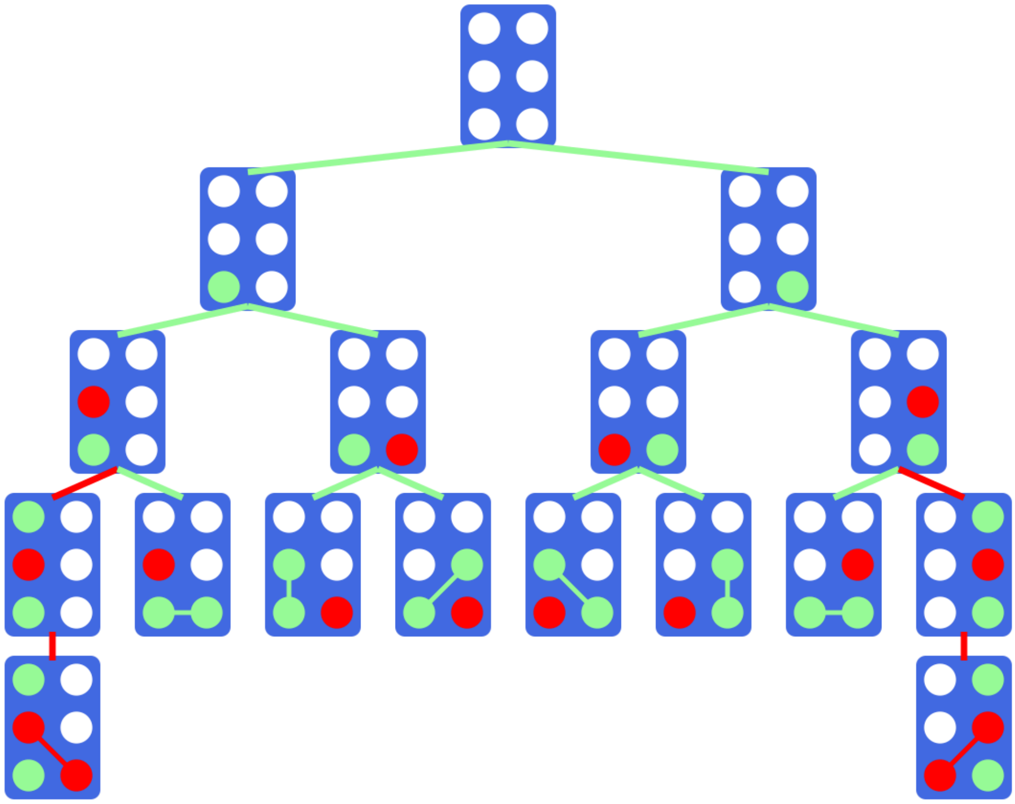
- Le Puissance 4 de 7 colonnes de hauteur 6 est résolu par ordinateur en 1988
- Utilisation de l'algorithme **minimax** vu au cours S4...
- ...mais avec l'aide de la programmation dynamique!

Puissance 2 de 2 colonnes de hauteur 2



```
minimax(2, 2, 2, scaling = 1.7)
```

Puissance 2 de 2 colonnes de hauteur 3



Puissance 2 : sans mémoïzation

Colonnes = 3

Hauteur = 3

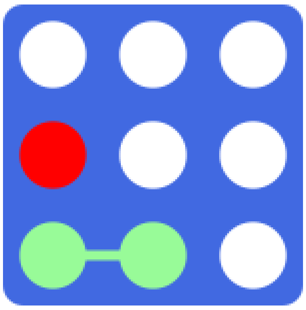
Longueur = 2

cols = [1, 1, 2]

```
cols = @time game(nrows, ncols, longueur)
```

```
  0.264744 seconds (281.06 k allocations: 14.031 MiB, 99.92% compilation time)
```

tour = 3



Mémoïsation pour Puissance 4

Colonnes = 3 Hauteur = 3

Observations clés:

- Arbre de recherche:
  - $\text{colonnes} = 3$ sous-branches par noeuds
  - profondeur : $\text{colonnes} \times \text{hauteur}$ donc ...
  - ... $3^{\text{colonnes} \times \text{hauteur}} = 19\,683$ feuilles !
- Stratégie optimale dépend de la **grille** mais **pas du passé**
- Chaque case a 3 états possibles: ,  et 
- $3^{\text{colonnes} \times \text{hauteur}} = 19\,683$ grilles différentes...
- ... mais beaucoup de ces grilles sont invalides ou **pas explorées**

Quelle **structure de donnée** utiliser pour la mémoïsation : état du jeu → action optimale ?

Mémoïsation pour Puissance 2

Colonnes = 3

Hauteur = 3

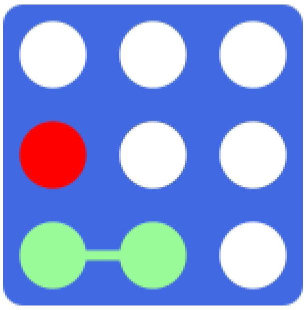
mem_cols = [1, 1, 2]

```
mem_cols = @time game(mem_nrows, mem_ncols, mem_longueur, memoization = true)
```

```
0.000040 seconds (18 allocations: 976 bytes)
```

Longueur = 2

tour = 3



Pièces de monnaie

Comme on a vu avec Fibonacci et le puissance 4, beaucoup de problème de calcul peuvent se représenter par un graphe où le but est de calculer la valeur d'un noeud du graphe et chaque arête (u, v) signifie que pour calculer la valeur du noeud u, il faut d'abord calculer la valeur du noeud v.

Considérons le problème de calculer de combien de manières différentes x_n, on sait faire une certaine somme avec des pièces de monnaie. Par exemple, pour faire 3 € avec les pièces de 1 € et 2 €, on peut soit utiliser 3 pièces de 1 € ou 1 pièce de 2 € et une pièce de 1 € donc il y a 2 possibilitées.

Quel graphe peut-on utiliser pour modéliser ce problème ? On peut utiliser un noeud par somme en €, représentant la solution du problème pour cette somme là. Si on ne travaille pas avec les cents, et que avec les billets jusque 10 €, on a

$$x_n = x_{n-10} + x_{n-5} + x_{n-2} + x_{n-1}$$

On a donc les arêtes $(n, n-10)$, $(n, n-5)$, $(n, n-5)$ et $(n, n-1)$.

▶ **Est-ce que cette formule donne le bon résultat ?**

Considérons la valeur $y_{n,k}$ qui correspond à la façon de faire la somme n avec des billets/pièces de valeur max k.

▶ **Quelle est la formule pour $y_{n,10}$? Quel est le graphe ? Est-ce que la bonne valeur est calculée ?**

Exercice: Implémenter le calcul avec l'approche top-down et bottom-up. Laquelle est la plus rapide ? Pourquoi ?

Supplémentaire: On peut aussi utiliser la formule

$$y_{n,10} = y_{n-10,5} + y_{n-20,5} + \dots + y_{n-5,2} + y_{n-10,2} + \dots$$

Modifier l'implémentation pour utiliser cette formule. Est-ce plus rapide ? Pourquoi ?

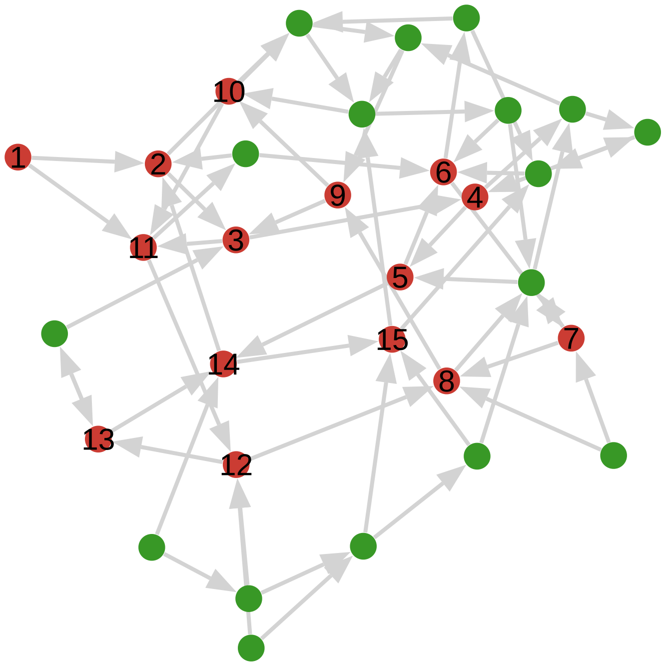
Depth-First Search (DFS)

```
dfs! (generic function with 1 method)
```

```julia
function dfs!(path, g::DiGraph, current, target)
    if current in path
        return false
    end
    push!(path, current)
    if current == target
        return true
    end
    for next in Graphs.outneighbors(g, current)
        if dfs!(path, g, next, target)
            return true
        end
    end
    pop!(path)
    return false
end
```

```
dfs (generic function with 1 method)
```

```julia
function dfs(g, current, target)
    path = Int[]
    dfs!(path, g, current, target)
    return path
end
```




```
let
    Random.seed!(32)
    n = 32
    g = random_regular_digraph(n, 2)
    path = dfs(g, 1, n)
    plot_path(g, path)
end
```

```
plot_path (generic function with 1 method)
```

```
function plot_path(g, path)
    cols = [colors.red, colors.green]
    gplot(g, nodefillc = [cols[v in path ? 1 : 2] for v in vertices(g)], nodelabel
    = [something(findfirst(isequal(v), path), "") for v in vertices(g)])
end
```

▶ **Quelle est la complexité de cet algorithme ?**

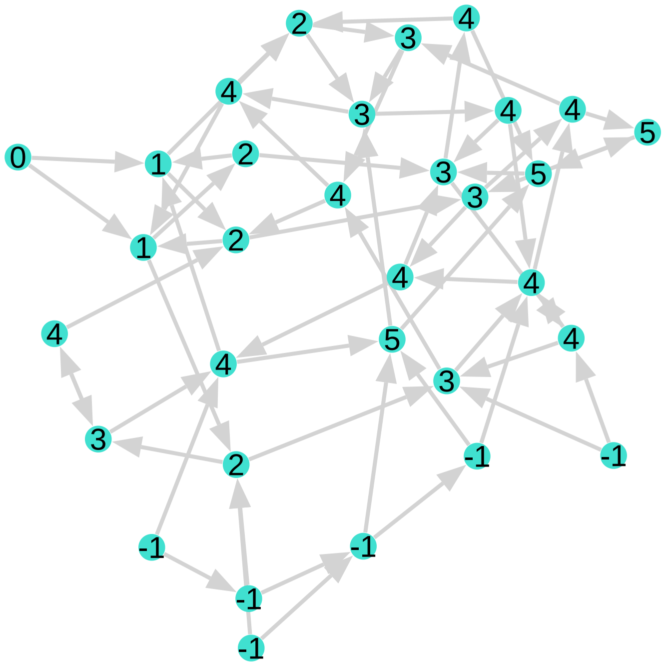
▶ **Trouve-t-il le plus court chemin ?**

Breadth-First Search (BFS)

L'algorithme BFS résoud le problème de shortest path dans un graphe dirigé dont tous les poids sont 1.

```
bfs (generic function with 1 method)
```

```julia
function bfs(g::DiGraph, source)
    dist = fill(-1, nv(g))
    queue = Queue{Tuple{Int,Int}}()
    dist[source] = 0
    enqueue!(queue, (source, 0))
    while !isempty(queue)
        current, d = dequeue!(queue)
        for next in outneighbors(g, current)
            if dist[next] == -1
                dist[next] = d + 1
                enqueue!(queue, (next, d + 1))
            end
        end
    end
    return dist
end
```




```
let
    Random.seed!(32)
    n = 32
    g = random_regular_digraph(n, 2)
    dist = bfs(g, 1)
    gplot(g, nodelabel = string.(dist))
end
```

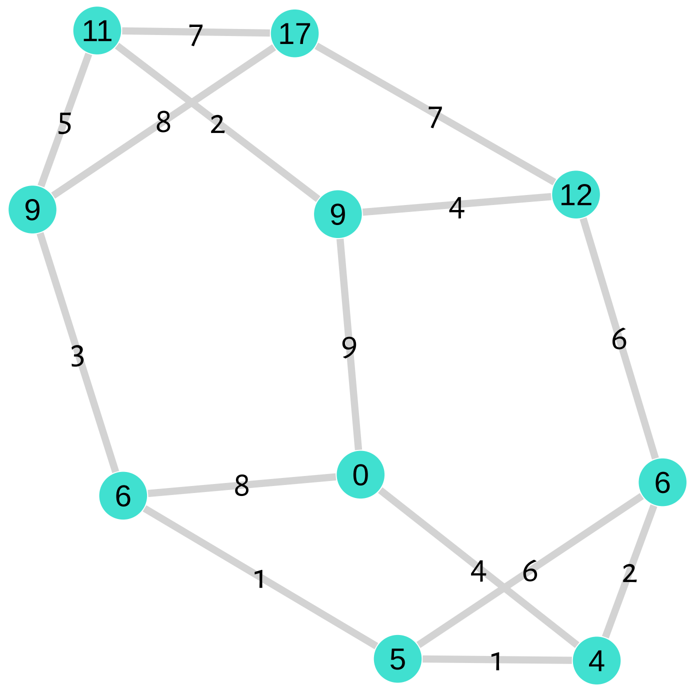
▶ **Quelle est la complexité de l'algorithme BFS ?**

Dijkstra

L'algorithme de Dijkstra résoud le problème de shortest path dans un graph dirigé quand tous les poids sont positifs.

```
dijkstra (generic function with 1 method)
```

```julia
function dijkstra(g::SimpleWeightedGraph, source)
    dist = fill(-1, nv(g))
    dist[source] = 0
    queue = PriorityQueue{Int,Int}()
    push!(queue, source => 0)
    while !isempty(queue)
        current = dequeue!(queue)
        for next in outneighbors(g, current)
            d = dist[current] + g[current, next, Val(:weight)]
            if dist[next] == -1
                dist[next] = d
                enqueue!(queue, next => d)
            elseif d < dist[next]
                dist[next] = d
                queue[next] = d
            end
        end
    end
    return dist
end
```




```julia
let
    Random.seed!(0)
    g = random_weighted(10, 3, 1:9)
    dist = dijkstra(g, 1)
    gplot(g, nodelabel=string.(dist), edgelabel=string.(Int.(weight.(edges(g)))))
end
```

▶ **Quelle est la complexité de l'algorithme Dijkstra ?**

▶ **Un fois qu'un noeud `v` est retourné par `dequeue!`, peut-il encore satisfaire `d < dist[next]` pour `v == next` par la suite ?**

▶ **Que se peut-il se passer si une des arêtes a un poids négatif ?**

Max Flow / Min Cut

- Le problème de Max Flow détermine le flot le plus important d'un noeud source à un noeud target étant donné une capacité maximale de chaque arête.
- Le problème de Min Cut détermine la somme des capacité minimale d'une coupe du graphe séparant la source de la target.

Théorème Pour tout graphe, le flot maximal est égal à la cut minimale !

Utils

```julia
using PlutoUI, Graphs, GraphPlot, Colors, DataStructures, SimpleWeightedGraphs
```

```
Precompiling Graphs...
   6492.7 ms  ✓ StaticArrays
    601.0 ms  ✓ StaticArrays → StaticArraysStatisticsExt
    949.7 ms  ✓ ArnoldiMethod
   3562.4 ms  ✓ Graphs
  4 dependencies successfully precompiled in 11 seconds. 26 already precompile
d.
Precompiling GraphPlot...
   2133.2 ms  ✓ Compose
   1201.2 ms  ✓ GraphPlot
  2 dependencies successfully precompiled in 3 seconds. 40 already precompile
d.
Precompiling SimpleWeightedGraphs...
    910.7 ms  ✓ SimpleWeightedGraphs
  1 dependency successfully precompiled in 1 seconds. 30 already precompiled.
```

```julia
import DocumenterCitations, Polyhedra, Makie, WGLMakie, StaticArrays, Bonito, Luxor
, Formatting, Random
```

```
Precompiling DocumenterCitations...
    492.6 ms  ✓ Expat_jll
    794.9 ms  ✓ BibParser
    551.1 ms  ✓ Git_jll
    420.4 ms  ✓ Git
   1305.0 ms  ✓ Bibliography
  25593.3 ms  ✓ Documenter
   3175.5 ms  ✓ DocumenterCitations
  7 dependencies successfully precompiled in 30 seconds. 61 already precompile
d.
Precompiling Polyhedra...
    713.9 ms  ✓ CodecBzip2
    823.2 ms  ✓ EarCut_jll
    876.3 ms  ✓ OpenSpecFun_jll
    914.8 ms  ✓ ConstructionBase → ConstructionBaseStaticArraysExt
   1015.6 ms  ✓ StructArrays → StructArraysStaticArraysExt
   1162.8 ms  ✓ BenchmarkTools
   2510.1 ms  ✓ SpecialFunctions
   5580.5 ms  ✓ GeometryBasics
   3380.3 ms  ✓ ForwardDiff
    682.8 ms  ✓ ForwardDiff → ForwardDiffStaticArraysExt
  85900.7 ms  ✓ MathOptInterface
  18244.9 ms  ✓ JuMP
   4131.1 ms  ✓ Polyhedra
  13 dependencies successfully precompiled in 115 seconds. 74 already precompi
led.
Precompiling Makie...
    745.7 ms  ✓ Rmath_jll
    799.7 ms  ✓ Graphite2_jll
```

qa (generic function with 2 methods)

```julia
include("utils.jl")
```

```julia
begin
    slider_n = @bind n Slider(1:42, default = 10, show_value = true);
    slider_k = @bind k Slider(1:42, default = 10, show_value = true);
    nothing
end
```

draw_fib (generic function with 1 method)

```
function draw_fib(n, depth)
    if n < 3
        return 0
    end
    dy = 140
    δ = round(Int, dy * 0.9^depth) * 1.2
    w = fib(n - 2)
    Δy = 30
    if depth != 1
        Luxor.translate(δ * w/2, 0)
    end
    if n == 3
        Luxor.setdash("dashed")
        Luxor.circle(Luxor.Point(6, -6), 20, :stroke)
    end
    Luxor.sethue("black")
    Luxor.text(string(n))
    Luxor.translate(-δ * w/2, 0)
    cur = 0
    for m in [n - 1, n - 2]
        if m < 3
            continue
        end
        dx = cur * δ
        Luxor.translate(dx, dy)
        dw = draw_fib(m, depth + 1)
        Luxor.sethue("black")
        Luxor.setdash("solid")
        Luxor.setline(δ/40)
        Luxor.line(Luxor.Point(-dx+δ * w/2, -dy+Δy), Luxor.Point(δ*dw/2, -Δy),
        :stroke)
        Luxor.translate(-dx, -dy)
        cur += dw
    end
    return w
end
```

draw_fibo (generic function with 1 method)

```
function draw_fibo(n)
    Luxor.@draw begin
        Luxor.translate(0, -350)
        Luxor.fontsize(30)
        Luxor.fontface("Alegreya Sans SC")
        draw_fib(n, 1)
    end 1200 800
end
```

random_weighted (generic function with 1 method)

```julia
function random_weighted(n, k, weights)
    g = random_regular_graph(n, k)
    wg = SimpleWeightedGraph(n)
    for e in edges(g)
        add_edge!(wg, src(e), dst(e), rand(weights))
    end
    return wg
end
```

colors = (red = , green = , blue = , purple =)

```julia
colors = Colors.JULIA_LOGO_COLORS
```

biblio =
CitationBibliography("/home/runner/work/LSINC1113/LSINC1113/Lectures/biblio.bib", AlphaSt

```julia
biblio = load_biblio!()
```

```
Loading bibliography from `/home/runner/work/LSINC1113/LSINC1113/Lectures/biblio.bib`...
```

```
Entry west2022Introduction is missing the publisher field(s).
```

```
Loading completed.
```

```
polyhedron =
Polyhedron Polyhedra.DefaultPolyhedron{Float64, Polyhedra.MixedMatHRep{Float64, Matrix{Fl
24-element iterator of Vector{Float64}:
 [0.0, 1.0, 2.0, 3.0]
 [0.0, 2.0, 1.0, 3.0]
 [1.0, 0.0, 2.0, 3.0]
 [1.0, 2.0, 0.0, 3.0]
 [2.0, 0.0, 1.0, 3.0]
 [2.0, 1.0, 0.0, 3.0]
 [0.0, 1.0, 3.0, 2.0]
 [0.0, 3.0, 1.0, 2.0]
 [1.0, 0.0, 3.0, 2.0]
 [1.0, 3.0, 0.0, 2.0]
 [3.0, 0.0, 1.0, 2.0]
 [3.0, 1.0, 0.0, 2.0]
 [0.0, 3.0, 2.0, 1.0]
 [0.0, 2.0, 3.0, 1.0]
  ⋮
```

```julia
polyhedron = Polyhedra.polyhedron(Polyhedra.vrep(Float64[
    0 1 2 3;0 2 1 3; 1 0 2 3; 1 2 0 3; 2 0 1 3; 2 1 0 3;
    0 1 3 2;0 3 1 2; 1 0 3 2; 1 3 0 2; 3 0 1 2; 3 1 0 2;
    0 3 2 1;0 2 3 1; 3 0 2 1; 3 2 0 1; 2 0 3 1; 2 3 0 1;
    3 1 2 0;3 2 1 0; 1 3 2 0; 1 2 3 0; 2 3 1 0; 2 1 3 0
]))
```

```
projected =
Polyhedron Polyhedra.DefaultPolyhedron{Float64, Polyhedra.MixedMatHRep{Float64, Matrix{Fl
24-element iterator of Vector{Float64}:
 [-0.7071067811865501, -1.2247448713915914, -1.7320508075688754]
 [0.7071067811865492, -1.2247448713915912, -1.7320508075688754]
 [-1.4142135623730974, -5.10702591327572e-15, -1.7320508075688759]
 [-0.7071067811865478, 1.224744871391588, -1.7320508075688759]
 [1.4142135623730963, -4.3298697960381105e-15, -1.7320508075688756]
 [0.7071067811865472, 1.2247448713915883, -1.7320508075688756]
 [-0.7071067811865465, -2.0412414523193156, -0.5773502691896244]
 [-2.121320343559643, 0.40824829046386146, -0.5773502691896253]
 [0.7071067811865529, -2.0412414523193156, -0.5773502691896242]
 [2.121320343559643, 0.40824829046385874, -0.5773502691896244]
 [-1.4142135623730931, 1.6329931618554543, -0.5773502691896262]
 [1.4142135623730934, 1.6329931618554512, -0.5773502691896251]
 [2.1213203435596464, -0.40824829046386274, 0.5773502691896222]
 [0.7071067811865501, 2.0412414523193165, 0.5773502691896217]
  ⋮
```

```julia
projected = Polyhedra.project(polyhedron, [1 1 1; -1 1 1; 0 -2 1; 0 0 -3])
```

```julia
struct Config{N}
    libre::StaticArrays.SVector{N,UInt8}
    player::StaticArrays.SVector{N,UInt8}
    max::UInt8
end
```

```
_copy (generic function with 3 methods)
```

```julia
begin
_num_rows(mat::AbstractMatrix) = size(mat, 1)
_num_rows(config::Config) = _shift_bit(config.max) - 1
_reverse(config::Config) = Config(reverse(config.libre), reverse(config.player), config.max)
Base.getindex(c::Config, i::UInt8, j::Int) = (!iszero(c.libre[j] & i), !iszero(c.player[j] & i))
_compare(player::Int, a, b) = player * a > player * b
_compare(player::Bool, a, b) = player ? a < b : a > b
_sign(player::Int, a) = player * a
_sign(w::Tuple{Bool,Bool}, a) = w[2] ? -a : a
_first_player(::AbstractMatrix) = 1
_first_player(::Config) = false
_next_player(player::Int) = -player
_next_player(player::Bool) = !player
_columns(m::AbstractMatrix) = axes(m, 2)
_columns(config::Config) = eachindex(config.libre)
function _next(mat::AbstractMatrix, col)
    for r in axes(config, 1)
        if iszero(config[r, col])
            return r
        end
    end
    return 0
end
function _next(config::Config, col)
    mask = config.libre[col] + 0b1
    if mask == config.max
        return 0b0
    end
    return mask
end
function _assign(mat::StaticArrays.SMatrix{I,J}, r, c, value) where {I,J}
    target = r + (c - 1) * I
    return StaticArrays.SMatrix{I,J}(ntuple(i -> i == target ? value : mat[i], Val(I * J)))
    #return @SMatrix [(row, col) == (r, c) ? value : mat[row, col] for row in 1:nrows, col in 1:ncols]
end
function _assign(v::StaticArrays.SVector{N}, mask, col) where {N}
    StaticArrays.SVector{N}(ntuple(c -> c == col ? (v[c] | mask) : v[c], Val(N)))
end
function _assign(config::Config, mask, col, value::Bool)
    libre = _assign(config.libre, mask, col)
    if value
        player = _assign(config.player, mask, col)
    else
        player = config.player
```

```
    end
    return Config(libre, player, config.max)
end
function _assign(mat::Matrix, row, col, value)
    mat[row, col] = value
    return mat
end
_unassign(::StaticArrays.SMatrix, _, _) = nothing
_unassign(::Config, _, _) = nothing
function _unassign(mat::Matrix, row, col)
    mat[row, col] = 0
    return
end
_copy(m::StaticArrays.SMatrix) = m
_copy(m::Config) = m
_copy(m::Matrix) = copy(m)
```

```
_has_winner (generic function with 2 methods)
```

```
begin
_no_winner(::AbstractMatrix) = 0
_no_winner(::Config) = (false, false)
_has_winner(x::Int) = !iszero(x)
_has_winner(x::Tuple{Bool,Bool}) = x[1]
end
```

```
const Δs = ▸[(-1, 0), (0, 1), (0, -1), (1, 1), (-1, 1), (1, -1), (-1, -1)]
```

```
const Δs = [(-1, 0), (0, 1), (0, -1), (1, 1), (-1, 1), (1, -1), (-1, -1)]
```

```
winner (generic function with 1 method)
```

```
function winner(config, longueur, pos)
    for Δ in Δs
        win = winner(config, pos, Δ, longueur)
        if _has_winner(win)
            return win, Δ
        end
    end
    return _no_winner(config), (0, 0)
end
```

winner (generic function with 2 methods)

```
function winner(config, longueur)
    for col in _columns(config)
        row = _first_row(config)
        while true
            pos = (row, col)
            win, Δ = winner(config, longueur, pos)
            if _has_winner(win)
                return win, pos, Δ
            end
            row = _next_row(row)
            if _done(config, row)
                break
            end
        end
    end
    return _no_winner(config), (0, 0), (0, 0)
end
```

_done (generic function with 2 methods)

```
begin
_first_row(::AbstractMatrix) = 1
_next_row(row::Int) = row + 1
_done(m::AbstractMatrix, row) = row > size(m, 1)
_first_row(::Config) = 0b1
_next_row(row::UInt8) = row << 1
_done(c::Config, row) = row == c.max
end
```

winner (generic function with 3 methods)

```
function winner(config, x, Δ::Tuple{Int,Int}, longueur)
    if !_inrows(config, x[1], (longueur::Int - 1) * Δ[1]) ||
        !in(x[2] + (longueur::Int - 1) * Δ[2], _columns(config))
    #if !all(in.(x .+ (longueur - 1) .* Δ, axes(config)))
        return _no_winner(config)
    end
    cur = config[x[1], x[2]]
    if iszero(cur[1])
        return _no_winner(config)
    end
    for i in 1:(longueur::Int - 1)
        if config[_shift(x[1], i * Δ[1]), _shift(x[2], i * Δ[2])] != cur
        #if config[(x .+ i .* Δ)...] != cur
            return _no_winner(config)
        end
    end
    return cur
end
```

```
_inrows (generic function with 2 methods)
```

```julia
begin
_shift(x::Int, y::Int) = x + y
_shift(x::UInt8, y::Int) = x << y
_inrows(m::AbstractMatrix, x::Int, y::Int) = in(x + y, axes(m, 1))
function _inrows(config::Config, x::UInt8, y::Int)
    if y == 0
        return true
    elseif y < 0
        return x >= (0b1 << (UInt8(-y)))
    else
        return (config.max >> UInt8(y)) > x
    end
end
end
```

```
play! (generic function with 1 method)
```

```julia
function play!(config, col::Int, player)
    row = _next(config, col)
    #row = @time findfirst(row -> iszero(config[row, col]), axes(config, 1))
    #if !isnothing(row)
    if !iszero(row)
        config = _assign(config, row, col, player)
    end
    return row, config
end
```

```
_draw (generic function with 2 methods)
```

```julia
function _draw(config, longueur, δ = 50)
    colors = Dict(
        (false, false) => "white", (true, true) => "red", (true, false) =>
        "palegreen",
        0 => "white", 1 => "palegreen", -1 => "red",
    )
    sz = (length(_columns(config)), _num_rows(config))
    c = (sz .+ 1) ./ 2 .* δ
    screen = sz .* δ
    pos(row, col) = Luxor.Point(δ * col, δ * (sz[2] - _shift_bit(row) + 1))
    Luxor.translate(-Luxor.Point(c...))
    Luxor.sethue("royalblue")
    Luxor.polysmooth(Luxor.box(Luxor.Point(c...), screen..., vertices=true), δ *
    0.2, :fill)
    for col in _columns(config)
        row = _first_row(config)
        while true
            Luxor.sethue(colors[config[row, col]])
            Luxor.circle(pos(row, col), δ / 3, :fill)
            row = _next_row(row)
            if _done(config, row)
                break
            end
        end
    end
    win, x, Δ = winner(config, longueur)
    if _has_winner(win)
        Luxor.sethue(colors[win])
        #polysmooth(Luxor.box(pos((x .+ Δ .* (longueur - 1) ./ 2)...), reverse((Δ
        .* δ .* (longueur - 0.4)) .+ δ/5)...), 4, :fill)
        Luxor.setline(δ/10)
        Luxor.line(pos(x...), pos((_shift_bit.(x) .+ Δ .* (longueur-1))...),
        :stroke)
    end
    Luxor.translate(Luxor.Point(c...))
end
```

```
_shift_bit (generic function with 2 methods)
```

```julia
begin
    _shift_bit(x::Int) = x
    function _shift_bit(x::UInt8)
        y = findfirst(i -> (0b1 << (i-1)) == x, 1:7)
        return y
    end
end
```

best_move_rec (generic function with 1 method)

```julia
function best_move_rec(config, player, longueur, depth, cache; patient, maxturn)
    if depth > maxturn + 1
        error("$depth > $maxturn + 1")
    end
    if cache !== nothing
        if haskey(cache, config)
            return convert(Tuple{Int,Int}, cache[config])
        end
        reversed = _reverse(config)
        if haskey(cache, reversed)
            return convert(Tuple{Int,Int}, cache[reversed])
        end
    end
    config = _copy(config)
    best = 0
    best_col = 0
    for col in _columns(config)
        row, new_config = play!(config, col, player)
        if !iszero(row)
            win, _ = winner(new_config, longueur, (row, col))
            if _has_winner(win)
                best = _sign(win, 2maxturn - depth)
                best_col = col
            end
        end
    end
    if iszero(best)
    for col in _columns(config)
        row, new_config = play!(config, col, player)
        if !iszero(row)
            cur, _ = best_move_rec(new_config, _next_player(player), longueur,
            depth + 1, cache; patient, maxturn)
            if best_col == 0 || _compare(player, cur, best)
                best = cur
                best_col = col
                if patient && _compare(player, best, 0)
                    break
                end
            end
            _unassign(config, row, col)
        end
    end
    end
    if cache !== nothing
        cache[config] = (best, best_col)
    end
    return best, best_col
end
```

game (generic function with 1 method)

```
game(nrows, ncols, longueur; memoization = false, patient = true) = game!
(_init(nrows, ncols), Int[], longueur, memoization; patient, maxturn = nrows *
ncols)
```

game! (generic function with 1 method)

```
function game!(config, cols, longueur, memoization; patient, maxturn)
    if memoization
        cache = Dict{typeof(config),Tuple{Int8,Int8}}()
    else
        cache = nothing
    end
    player = _first_player(config)
    for i in 1:maxturn
        _, col = best_move_rec(config, player, longueur, 1, cache; patient, maxturn)
        if col == 0
            break
        end
        push!(cols, col)
        _, config = play!(config, col, player)
        w = winner(config, longueur)
        if _has_winner(w[1])
            break
        end
        player = _next_player(player)
    end
    return cols
end
```

draw (generic function with 2 methods)

```
function draw(config, longueur, δ = 50)
    sz = (length(_columns(config)), _num_rows(config))
    screen = sz .* δ
    Luxor.@draw begin
        _draw(config, longueur, δ)
    end screen[1] screen[2]
end
```

play (generic function with 1 method)

```
function play(nrows, ncols, cols)
    config = _init(nrows, ncols)
    player = _first_player(config)
    for col in cols
        _, config = play!(config, col, player)
        player = _next_player(player)
    end
    return config
end
```

```
_init (generic function with 1 method)
```

```julia
function _init(nrows, ncols)
    z = StaticArrays.SVector{ncols}(ntuple(_ -> zero(UInt8), Val(ncols)))
    Config(z, z, (0b1 << nrows))
end
```

```
minimax (generic function with 1 method)
```

```julia
function minimax(nrows::Integer, ncols, longueur; vert = -200, w = 1200, h = 600,
dy = 140, damping = 0.9, Δy = 55, scaling)
    config = _init(nrows, ncols)
    Luxor.@draw minimax(config, _first_player(config), longueur, 1; dy, damping,
    vert, Δy, scaling) w h
end
```

```julia
function width(config, player, longueur, damping, depth)
    if _has_winner(winner(config, longueur)[1])
        return damping^depth
    end
    w = 0
    for col in _columns(config)
        row, new_config = play!(config, col, player)
        if !iszero(row)
            w += width(new_config, _next_player(player), longueur, damping, depth +
            1)
        end
    end
    return max(w, 1)
end;
```

```julia
function minimax(config, player, longueur, depth; dy, damping, vert, Δy, scaling)
    δ = round(Int, dy * damping^depth) * scaling
    w = width(config, player, longueur, damping, depth)
    if all(iszero(config.libre))
        Luxor.translate(0, vert)
    else
        Luxor.translate(δ * w/2, 0)
    end
    _draw(config, longueur)
    Luxor.translate(-δ * w/2, 0)
    if _has_winner(winner(config, longueur)[1])
        return w
    end
    cur = 0
    for col in _columns(config)
        row, new_config = play!(config, col, player)
        if !iszero(row)
            dx = cur * δ
            Luxor.translate(dx, dy)
            win, _, _ = winner(new_config, longueur)
            score, _ = best_move_rec(new_config, _next_player(player), longueur,
            depth + 1, nothing, patient = true, maxturn = 30)
            dw = minimax(new_config, _next_player(player), longueur, depth + 1; dy,
            damping, vert, Δy, scaling)
            if _has_winner(win)
                if player
                    Luxor.sethue("red")
                else
                    Luxor.sethue("palegreen")
                end
            elseif score == 0
                Luxor.sethue("black")
            elseif score > 0
                Luxor.sethue("palegreen")
            else
                Luxor.sethue("red")
            end
            Luxor.setline(δ/20)
            Luxor.line(Luxor.Point(-dx+δ * w/2, -dy+Δy), Luxor.Point(δ*dw/2, -Δy),
            :stroke)
            Luxor.translate(-dx, -dy)
            cur += dw
        end
    end
    return w
end;
```

mesh =

▶ Mesh(Polyhedron Polyhedra.DefaultPolyhedron{Float64, Polyhedra.MixedMatHRep{Float64, Ma
    24-element iterator of Vector{Float64}:

```
mesh = Polyhedra.Mesh(projected)
```

cite (generic function with 1 method)

```
cite(args...) = bibcite(biblio, args...)
```

refs (generic function with 1 method)

```
refs(args...) = bibrefs(biblio, args...)
```