1.

```ocaml
type order = bool array array

let check_reflexivity r =
    let n = Array.length r in
    let ans = ref true in
    for i = 0 to n - 1 do
        if not r.(i).(i) then ans := false
    done;
    !ans
```

2.

```ocaml
let check_reflexivity_bis r =
    let n = Array.length r in
    let rec aux i =
        if i = n then true
        else if not r.(i).(i) then false
        else aux (i + 1) in
    aux 0
```

3.

```ocaml
let check_antisymmetry r =
    let n = Array.length r in
    let ans = ref true in
    for i = 0 to n - 1 do
        for j = 0 to n - 1 do
            if r.(i).(j) && r.(j).(i) && i <> j then ans := false
        done
    done;
    !ans
```

4.

```ocaml
let check_transitivity r =
    let n = Array.length r in
    let ans = ref true in
    for i = 0 to n - 1 do
        for j = 0 to n - 1 do
            for k = 0 to n - 1 do
                if r.(i).(j) && r.(j).(k) && not r.(i).(k) then ans := false
            done
        done
    done;
    !ans
```

5.

```ocaml
let is_order r =
    check_reflexivity r && check_antisymmetry r && check_transitivity r
```

6.

```ocaml
let outdegrees r =
    let n = Array.length r in
    let outd = Array.make n 0 in
    for i = 0 to n - 1 do
        for j = 0 to n - 1 do
            if r.(i).(j) then outd.(i) <- outd.(i) + 1
        done
    done;
    outd
```

7.

```
let argmin d =
    let n = Array.length d in
    let ans = ref 0 in
    for i = 1 to n - 1 do
        if d.(i) < d.(!ans) then ans := i
    done;
    !ans
```

8. Le dernier sommet v d'un tri topologique est de degré sortant égal à 1 (il y a seulement un arc (v, v)) : s'il y avait une arc (v, w) avec $w \neq v$ alors on aurait $\tau(v) < \tau(w)$ ce qui est faux car v est le dernier sommet.

9.

```
let topological_sort r =
    let n = Array.length r in
    let outd = outdegrees r in
    let ans = Array.make n 0 in
    for i = n - 1 downto 0 do
        let v = argmin outd in
        outd.(v) <- max_int;
        for j = 0 to n - 1 do
            if r.(v).(j) then outd.(j) <- outd.(j) - 1
        done;
        ans.(i) <- v
    done;
    ans
```

10. Invariant de boucle : à la fin de l'itération i, les sommets de `ans.(i)` à `ans.(n - 1)` sont triés topologiquement.

11. Complexité $O(n^2)$, plus élevé que l'inverse d'un parcours postfixe (voir cours) en $O(n + p)$.

12. On montre par récurrence sur k que si `c` contient k éléments alors `is_chain r c` renvoie vrai si et seulement si `c` est une chaîne de r.

13. `is_chain r c` exécute `for_all` sur une liste de taille γ, puis $\gamma - 1$, etc. jusqu'à 1. La complexité est donc $\sum_{i=1}^{\gamma} i = \frac{\gamma(\gamma + 1)}{2} = O(\gamma^2)$.

14. <u>Solution 1</u> : On utilise un tas min avec la relation $\trianglelefteq$ à la place d'une priorité : si u est le père de v dans le tas, on doit avoir $u \trianglelefteq v$.
À chaque fois que l'on extrait la racine du tas, celle-ci est inférieure (pour $\trianglelefteq$) à toutes les autres valeurs du tas, par transitivité de $\trianglelefteq$.
On renvoie `false` si on trouve un sommet qui n'est pas comparable avec son père (lors d'un ajout ou d'une extraction).
Comme toutes les opérations de tas sont en $O(\log \gamma)$ (car un tas est de hauteur logarithmique en le nombre d'éléments), la complexité est bien $O(\gamma \log \gamma)$.
<u>Solution 2</u> : Utiliser un arbre binaire de recherche équilibré pour avoir l'insertion et suppression en $O(\log(\gamma))$ (exemple : arbre rouge-noir) avec la relation $\trianglelefteq$. Si l'arbre est correctement formé, deux éléments quelconques u et v sont forcément comparables, en utilisant le chemin de u à v dans l'arbre et la transitivité de $\trianglelefteq$.
`push q e` renvoie `false` si, l'insertion de `e` (comme dans un ABR classique), on trouve un sommet avec lequel `e` n'est pas comparable.
`pop q` supprime un élément (par exemple le minimum qui est celui tout à gauche) et renvoie toujours `true`.

15.

```
let rec is_antichain r a = match a with
    | [] -> true
    | v::q -> List.for_all (fun u -> not r.(u).(v) && not r.(v).(u)) q && is_antichain r q
```

Complexité en $O(\gamma^2)$.

16. Supposons qu'il existe un algorithme en complexité $o(\gamma^2)$ pour vérifier si un ensemble est une antichaîne. Comme il y a

$\dfrac{\gamma(\gamma-1)}{2}$ paires de sommets, il y a donc, si γ est assez grand, un couple (x, y) qui n'est pas testé par l'algorithme. Pour cette valeur de γ, considérons alors les ordres :

- r_1 tel que $\forall u, v,\ r_1(u, v) =$ `true` si $u = v$ et `false` sinon.
- r_2 tel que $r_2(x, y) =$ `true` et $\forall (u, v) \neq (x, y),\ r_2(u, v) = r_1(u, v)$.

L'algorithme renvoie alors la même valeur pour r_1 et r_2 (contrairement à la vérification d'une chaîne, on ne peut pas utiliser la transitivité pour réduire le nombre de comparaisons). On a donc une contradiction.

17. A ne peut pas contenir 2 éléments du même C_ℓ car ils seraient comparables. On a donc $|A| \leq \lambda$.

18.

```
type bipartite = int list array

let bipartite_of_order r =
    let n = Array.length r in
    let g = Array.make (2*n) [] in
    for i = 0 to n - 1 do
        for j = 0 to n - 1 do
            if r.(i).(j) then (
                g.(i) <- (j + n)::g.(i);
                g.(j + n) <- i::g.(j + n)
            )
        done
    done;
    g
```

19. Supposons par l'absurde que C_k ne soit pas une chaîne et soient $u, v \in C_k$ incomparables ($u \ntrianglelefteq v$ et $u \ntrianglelefteq v$).
Soit $C = \{w \in V \mid \exists\ (v, w_{2\times 2}), (w_2, w_{2\times 3}), ..., (w_{p-1}, w) \in M\} \cup \{w \in V \mid \exists\ (w, w_{2\times 2}), (w_2, w_{2\times 3}), ..., (w_{p-1}, v) \in M\}$ l'ensemble des sommets atteignable depuis v et ceux permettant d'atteindre v, en utilisant des arêtes de M.
En remplaçant C_k par $C_k \setminus C$ et C, on obtient une partition de V qui vérifie 1. mais qui est de cardinal strictement supérieur à λ, ce qui est absurde d'après 2..
Remarque : ceci montre aussi que chaque C_k est un chemin constitué d'arêtes de M.

20. On peut utiliser une structure d'Union-Find dont chaque ensemble est une chaîne :

```
let chains_of_matching m =
    let uf = Array.init n (fun i -> i) in
    for i = 0 to n - 1 do
        if m.(i) <> -1 then (
            let u = find uf i in
            let v = find uf m.(i) in
            if u <> v then uf.(u) <- v
        )
    done;
    let chains = Array.make n [] in
    for i = 0 to n - 1 do
        chains.(find uf i) <- i::chains.(find uf i)
    done;
    let rec to_list i =
        if i = n then []
        else if chains.(i) = [] then to_list (i + 1)
        else chains.(i)::to_list (i + 1) in
    to_list 0
```

21. Soit $l \in [\![1, \lambda]\!]$. On considère le minimum v_{i_1} de C_ℓ pour l'ordre $\trianglelefteq$ (c'est-à-dire tel que $\forall w \in C_\ell, w \neq v_{i_1} \implies v_{i_1} \trianglelefteq w$)
Pour montrer l'existence d'un minimum, on peut prendre un élément quelconque $u_1 \in C_\ell$: soit c'est un minimum, soit il existe un élément $u_2 \in C_\ell$ tel que $u_2 \neq u_1$ et $u_2 \trianglelefteq u_1$. On peut alors réitérer le processus pour obtenir un minimum (sinon on tombe sur un cycle qui donne $u_1 = u_2 = ...$ ce qui est absurde).
On définit ensuite v_{i_2} comme le minimum de $C_\ell \setminus \{v_{i_1}\}$ pour l'ordre $\trianglelefteq$, et ainsi de suite jusqu'à v_{i_l}.
L'ensemble des arêtes $\{w_{i_1}, w_{2i_2}\}, \{w_{i_2}, w_{2i_3}\}, ..., \{w_{i_{l-1}}, 2w_{i_l}\}$ est un couplage de P qui donne C_ℓ.
On prend alors comme couplage M l'union de ces couplages pour chaque C_ℓ.

22. Dans la construction de la question précédente, le couplage obtenu dans chaque C_ℓ est de taille $|C_\ell| - 1$. Au total, M est donc de taille $|V| - \lambda$. Maximiser $|M|$ revient donc à minimiser λ.

23. Voir cours.

24. D'après 23 :

```
let minimum_cover r =
    r |> bipartite_of_order |> maximum_matching |> chains_of_matching
```