Partie 1

1.1 2 déclarations dont on peut vérifier la véracité suffisent, ce sera VV, FF en alternance.
(et une ⇒ ambiguïté)

1.2 $A_V = \bigwedge_{i=1}^{n} A_i$, $A_M = \bigwedge_{i=1}^{n} \neg A_i$, $A_{CV} = \bigwedge_{\substack{i=1 \\ i \text{ impair}}}^{n} A_i \wedge \bigwedge_{\substack{i=2 \\ i \text{ pair}}}^{n} \neg A_i$, $A_{CM} = \bigwedge_{\substack{i=1 \\ i \text{ impair}}}^{n} \neg A_i \wedge \bigwedge_{\substack{i=2 \\ i \text{ pair}}}^{n} A_i$

1.3 $A_1 = R \wedge \neg B$, $A_2 = R \wedge \neg V \vee \neg R \wedge V$, $A_3 = (R \wedge V) \Rightarrow B$

1.4 $A_V = (R \wedge \neg B) \wedge (R \oplus V) \wedge ((R \wedge V) \Rightarrow B)$

$A_M = (\neg R \vee B) \wedge (R \Leftrightarrow V) \wedge (R \wedge V \wedge \neg B)$

$A_{CV} = (R \wedge \neg B) \wedge (R \Leftrightarrow V) \wedge ((R \wedge V) \Rightarrow B)$

$A_{CM} = (\neg R \vee B) \wedge (R \oplus V) \wedge (R \wedge V \wedge \neg B)$

1.5

R	V	B	A_1	A_2	A_3	A_V	A_M	A_{CV}	A_{CM}
✗	✗	✗	✗	✗	✓	✗			
✗	✗	✓	✗	✗	✓	✗			
✗	✓	✗	✗	✓	✓	✗			
✗	✓	✓	✗	✓	✓	✗			
✓	✗	✗	✓	✓	✓	✓			
✓	✗	✓	✗	✓	✓	✗			
✓	✓	✗	✓	✗	✗	✗			
✓	✓	✓	✗	✗	✓	✗			

L'orateur est un véridique qui n'aime que le rouge.

1.6 $G_1 = C \Rightarrow L$, $H_1 = C \wedge \neg T \vee \neg C \wedge T$, $I_1 = L$, $I_2 = \neg T$

1.7 $\neg G_1 \wedge H_1 \wedge (I_1 \wedge \neg I_2 \vee \neg I_1 \wedge I_2)$

1.8 $(C \wedge \neg L) \wedge (C \oplus T) \wedge (L \wedge T \vee \neg L \wedge \neg T)$

$\Leftrightarrow C \wedge \neg L \wedge \neg T$ donc seul le cercle est visible et I commence par mentir.

Partie 2

2.1 Soient s = "aa" et t = "aaa", alors 2 et 3 sont deux occurrences de s dans t avec $2 \leq 3$ et $2 \geq 3-2+1$.

2.2 La première occurrence est $\geq k$, la dernière est $\leq n$, il y en a donc au plus $n-k+1$, borne atteinte avec des mots avec une seule lettre distincte, et la même.

2.3
```
let rec longueur = function |[] -> 0 |_::q -> 1 + longueur q;;
```
$O(n)$ évident.

2.4
```
let rec prefixe s t = match s,t with |[], _ -> true |_, [] -> false |a::q, b::qq -> a=b && prefixe q qq;;
```
$O(\min(k,n))$ subtil ! k accepté

2.5
```
let recherche_naive s t = let k = longueur s in
  let rec tous_tests i = function
    |[] -> []
    |b::qq when prefixe s (b::qq) -> (i+k) :: tous_tests (i+1) qq
    |_::qq -> tous_tests (i+1) qq
  in tous_tests 0 t;;
```

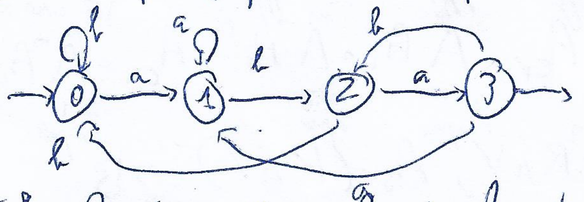
2.6 $O(k\,n)$ $\boxed{\text{+k subtil pour gérer le cas où } n=0}$

nombre d'appels à tous_tests

coût du 2e when en raison de l'appel à prefixe

2.7 $(\rho^j(q))_{j \in \mathbb{N}}$ est une suite strictement décroissante jusqu'à atteindre 0 (donc forcément atteint).
Soit $j \geq 0$ tel que $\rho^j(q) = 0$. Par hypothèse, $\delta(0, \alpha)$ est défini, CQFD.

2.8



2.9 $\Sigma^* aba$ (se voit en lisant aba depuis n'importe quel état, en étudiant la réciproque).

2.10

```
let copie_afdr a = let fc = Array.copy a.final and rc = Array.copy a.repli
and tc = Array.copy a.transition in for i = 0 to Array.length tc - 1 do
tc.(i) <- Array.copy a.transition.(i) done; {final = fc; transition = tc; repli = rc};;
```

2.11
$O(k)$

```
let transition etat lt tc rc =
  let q = ref etat in while tc.(!q).(lt) = -1 do q := rc.(!q) done; tc.(!q).(lt);;
```

$O(k \times \lambda)$ pour la recopie, merde, c'est $O(k^2 \lambda)$ (*)

```
let enleve_repli a = let ac = copie_afdr a in let k = Array.length ac.transition
and lam = Array.length ac.transition.(0) in for i = 0 to k-1 do for j = 0 to lam-1 do
if ac.transition.(i).(j) = -1 then ac.transition.(i).(j) <- transition i j ac.transition ac.repli
done done; for i = 0 to k-1 do ac.repli.(i) <- i done; ac;;
```

2.12 Lancer (récursivement, juste pour le principe) la lecture du mot sur l'automate et relever à chaque fois si on est dans un état final. La récursion porte sur le mot, un entier suivant l'indice en cours et l'état en cours.

2.13

```
let occurrences a mot =
  let rec aux etat i = function
  | [] -> ~~if a.final.(etat) then [i] else []~~ []
  | lt :: q -> let etat2 = a.transition.(etat).(lt) in let l = aux etat2 (i+1) q in if a.final.(etat2)
      then (i+1) :: l else l
  in aux 0 0 mot;;
```

$O(n)$ peu importent k et λ.

(*) En vrai, les transitions étant mises à jour état par état, la fonction transition fait alors toujours au plus un tour dans chaque boucle while, donc on a quand même la bonne complexité.

Partie 3

0 ~~1~~ 2 3 ~~4~~ 5 ~~6~~ 7 8 9 | → Produit divisible par 6
→ Au moins un chiffre ≤ 4

Si 6 dans le nombre, 6 divise le produit de ses chiffres, absurde !
Si 1 dans le nombre, le produit des chiffres est impair, or il est divisible par 6, absurde !
Si 4 dans le nombre, 5 n'est pas dedans, donc le nombre a plus de 6 chiffres, donc au moins une répétition, donc pas de 3, donc un 0 pour que le produit soit divisible par 6, donc pas de 2 sinon le 0 serait en tête, ce qui a été interdit (au moins), mais avec seulement des 0 et des 4 la somme ne peut pas être un nombre premier, absurde !
Si 8 dans le nombre, 7 et 9 ne sont pas dedans, et 2 et 3 ne peuvent pas y être simultanément. Comme on sait déjà que 2 exclut 0, 2 est interdit car le produit ne peut pas être divisible par 6, donc 8 peut cohabiter avec 0, 3 et 5, et sans 5 il ne peut pas y avoir de 3, puis l'argument ayant forcément exclu 4 s'applique, donc il faut au moins un 5. Comme il n'y a pas de 9 on ne peut pas avoir 3, 5 et 8, donc les chiffres sont 0, 5 et 8 avec une répétition (absence de 3) mais forcément de 0 (sinon $0 + x = x$ contredit l'absence de 9), le 5 est à la fin (absence de 7), le 8 au début (0 ne peut pas y être), la somme fait bien 13 et on peut proposer $\boxed{8005}$ mais pas 80005 par exemple car $0 + 0 = 0$.
Compléter le raisonnement : Si 8 pas dans le nombre, on a forcément 2 et 3 ensemble, 0 est exclu, 5 obligatoire sinon on n'a pas assez de chiffres, 9 obligatoire car $2+3=5$ et les possibilités sont 2~~35~~9 (Σ (impair)) ou 2~~53~~9 (somme non première)
d'où l'unicité.