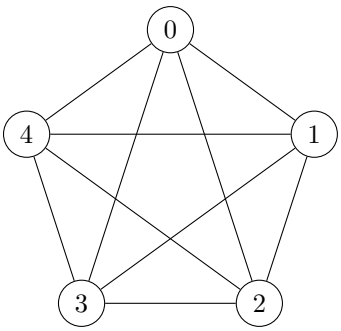
2.(b)

```
let diam_max n =
  let g = Array.make n [] in
  for i = 1 to n - 2 do
    g.(i) <- [i-1; i+1]
  done;
  if n > 1 then (g.(0) <- [1]; g.(n-1) <- [n-2]);
  g;;
```

3. (a) Un graphe complet (où toutes les arêtes possibles sont présentes) donne un diamètre de 1, qui est bien minimum:



(b)

```
let diam_min n =
  let g = Array.make n [] in
  for i = 0 to n - 1 do
    for j = 0 to n - 1 do
      if i <> j then g.(i) <- j::g.(i)
    done
  done;
  g;;
```

4. **Entrée**: un graphe G (orienté ou non) pondéré dont tous les poids sont positifs et un sommet s de G.
Sortie: la distance pondérée de s à chaque sommet de G (par exemple sous forme d'un tableau).
Étant donné un graphe non pondéré, on peut mettre un poids de 1 sur chaque arête et la distance pondérée est alors égale à la distance définie par l'énoncé. Il suffit ensuite d'appliquer une fois l'algorithme de Dijkstra depuis chaque sommet du graphe: la distance maximum trouvée est alors le diamètre.

5. On peut utiliser un parcours en largeur depuis chaque sommet du graphe, ce qui permet aussi d'obtenir toutes les distances donc le diamètre.

6. On sait d'après le cours que la complexité de l'algorithme de Dijkstra est plus élevée que celle du parcours en largeur. Comme dans les deux cas on applique l'algorithme autant de fois qu'il y a de sommets, il est préférable d'utiliser des parcours en largeur.
Plus précisément, si G est un graphe représenté par liste d'adjacence avec n sommets et m arêtes alors un parcours en largeur est en $O(n+m)$ donc la méthode de la question 5 est en $O(n(n+m))$ alors qu'une application de l'algorithme de Dijkstra est en $O(m\log(n))$ – si l'on utilise une file de priorité avec ajout et mise à jour en complexité logarithmique – ce qui donne une complexité totale $O(mn\log(n))$.

7.
```
Noeud (0, Noeud (1, Noeud (2, Noeud (4, Feuille, Feuille), Feuille),
                   Noeud (3, Noeud (5, Feuille, Feuille), Noeud (6, Feuille, Feuille))),
          Feuille)
```
Le diamètre de $G_{\mathcal{A}}$ est 4 et ses chemins maximaux sont $4,2,1,3,6$ et $4,2,1,3,5$

8. Soit $\mathcal{A}$ un arbre binaire enraciné en s et dont l'ensemble des noeuds est N. La fonction qui à chaque noeud v de $N\backslash\{s\}$ associe l'arc aboutissant en v est une bijection de $N\backslash\{s\}$ vers l'ensemble des arcs de $\mathcal{A}$.
On en déduit donc que $\boxed{r = n-1}$.
Remarque: on peut aussi démontrer ce résultat par récurrence (sur le nombre de noeuds, par exemple).

9.

```
let rec nb_noeuds a = match a with
  | Feuille -> 0
  | Noeud(r, g, d) -> 1 + nb_noeuds g + nb_noeuds d;;
```

10.

```
let numerotation a =
  let compteur = ref (-1) in
  let rec aux = function
    | Feuille -> Feuille
    | Noeud(_, g, d) -> (incr compteur;
                         let r = !compteur in
                         Noeud(r, aux g, aux d)) in
  aux a;;
```

Remarque technique: écrire `Noeud(!compteur, aux g, aux d)` ci-dessus ne fonctionnerait pas car les arguments sont évalués de droite à gauche (essayer `(print_int 1, print_int 2);;` par exemple, même si cela peut dépendre de la version de Caml). Ainsi `aux d` et `aux g` seraient d'abord évalués (ce qui modifie `compteur`) donc la valeur de `!compteur` ne serait plus la bonne.

11. On parcourt l'arbre en reliant chaque noeud avec son père:

```
let arbre_vers_graphe a =
  let ga = Array.make (nb_noeuds a) [] in
  let rec aux p = function (* p est le père de r *)
    | Feuille -> ()
    | Noeud(r, g, d) -> (if p <> -1 then (ga.(r) <- p::ga.(r);
                                          ga.(p) <- r::ga.(p));
                         aux r g;
                         aux r d) in
  aux (-1) a; (* -1 pour indiquer que la racine n'a pas de père *)
  ga;;
```

12. On peut numéroter les n sommets de l'arbre avec `numerotation`, le transformer en graphe avec `arbre_vers_graphe`, puis calculer son diamètre en utilisant la question 5.
`numerotation` et `arbre_vers_graphe` parcourent chaque sommet de l'arbre une fois en faisant un nombre constant d'itérations, donc sont en $O(n)$. Comme le nombre d'arête de l'arbre est $n-1$, d'après la réponse à la question 20, la méthode pour calculer le diamètre dans le graphe obtenu est en $O(n^2)$.
D'où la complexité totale $O(n) + O(n) + O(n^2) = \boxed{O(n^2)}$.

13. On note $\|C\|$ la longueur d'un chemin C. Si C est vide (c'est à dire qu'il ne contient aucun sommet) on définit $\|C\| = -1$. On pose aussi $h(\text{Feuille}) = 0$ (non défini par l'énoncé).
Soit $\mathcal{A}$ un arbre et C un chemin maximal de $G_{\mathcal{A}}$. Supposons que C passe par la racine de $\mathcal{A}$.
Soit C_g la partie de C dans $G_{\mathcal{A}_g}$ et $\overrightarrow{C_g}$ le chemin correspondant dans $\mathcal{A}_g$. Montrons que $\|C_g\| = h(\mathcal{A}_g) - 1$.
$\overrightarrow{C_g}$ est un plus long chemin de la racine de $\mathcal{A}_g$ à un noeud de $\mathcal{A}_g$ (sinon, on pourrait remplacer C_g dans C par un chemin plus long, ce qui contredirait la maximalité de C).
Donc $\|C_g\| = \|\overrightarrow{C_g}\| = h(\mathcal{A}_g) - 1$. Remarquons que cette formule reste vraie si $\mathcal{A}_g$ est une feuille.
On raisonne de même pour la partie C_d de C dans $G_{\mathcal{A}_d}$, d'où $\|C\| = \|C_g\| + 2 + \|C_d\| = \boxed{h(\mathcal{A}_g) + h(\mathcal{A}_d)}$ (on rajoute 2 pour les arêtes sortantes de la racine de $\mathcal{A}$).

14. Le diamètre est obtenu récursivement en remarquant qu'un chemin de longueur maximum est soit entièrement dans $\mathcal{A}_g$ (donc de longueur égale au diamètre de $\mathcal{A}_g$), soit entièrement dans $\mathcal{A}_d$ (donc de longueur égale au diamètre de $\mathcal{A}_d$) soit passe par la racine (donc de longueur $h(\mathcal{A}_g) + h(\mathcal{A}_d)$, d'après la question précédente).
On a besoin à la fois du diamètre et de la hauteur dans cette formule de récurrence, il est donc judicieux d'utiliser une fonction auxiliaire qui renvoie les deux informations:

```
let diam_arbre arb =
  let rec aux a = match a with (* renvoie (diamètre de a, hauteur de a) *)
    | Feuille -> (-1, 0)
    | Noeud(_, g, d) -> let dg, hg = aux g in
                        let dd, hd = aux d in
                        (max (max dg dd) (hg + hd), 1 + max hg hd) in
  fst (aux arb);;
```

On choisit de renvoyer $(-1, 0)$ pour un arbre réduit à une feuille en accord avec les conventions utilisées dans la réponse à la question 13.
`aux a` effectue un appel récursif pour chaque noeud de `a` et chacun de ces appels effectue un nombre constant d'opérations (en dehors des appels récursifs) donc la complexité de cette fonction est bien linéaire en le nombre de noeuds.
Remarque: ce ne serait pas le cas si on appelait une fonction recalculant la hauteur à chaque fois (on aurait alors une complexité quadratique).