Solutions partielles des exercices de révision pour l'examen final.

1. 5 6 4 3

3. Voici deux versions, l'une itérative, l'autre récursive

```
public static int nbMaillons ( Maillon debut ) {
    int reponse = 0;
    while ( debut != null ) {
        ++reponse;
        debut = debut.suivant();
    }
    return reponse;
}

public static int nbMaillons ( Maillon debut ) {
    int reponse = 0;
    if ( debut != null ) {
        reponse = 1 + nbMaillons ( debut.suivant() );
               // 1 + le nombre de maillon dans le reste de la liste
    }
    return reponse;
}
```

4.

```
public static Maillon copieInverse ( Maillon debut ) {
    Maillon reponse = null;
    while ( debut != null ) {
        // placer un nouveau maillon au début de reponse
        reponse = new Maillon ( debut.info (), reponse );
        debut = debut.suivant();
    }
    return reponse;
}
```

5. Voici trois versions, deux itératives, l'autre récursive

```
  public static Maillon copieEnOrdre ( Maillon debut ) {
      // on peut utiliser debut comme variable de travail puisqu'il s'agit d'une copie
      Maillon reponse = null;
      Maillon dernier = null;
      if ( debut != null ) { // au moins 1 maillon
          reponse = new Maillon ( debut.info() );
          dernier = reponse;
          debut = debut.suivant();
          while ( debut != null ) {
              dernier.modifierSuivant ( new Maillon ( debut.info() ) );
              dernier = dernier.suivant();
              debut = debut.suivant();
          }
      }
      return reponse;
  }

  public static Maillon copieEnOrdre ( Maillon debut ) {
      Maillon reponse = null;
      if ( debut != null ) {
          reponse = new Maillon ( debut.info() );
                                  // créer un nouveau maillon avec le premier élément
          reponse.modifierSuivant ( copieEnOrdre ( debut.suivant() ) );
                                  // copier le reste de la liste
      }
      return reponse;
  }
```

```
    public static Maillon copieEnOrdreVersion2 ( Maillon debut ) {
    // Version qui réutilise le code déjà écrit.
    // Utilise plus d'espace memoire (qui sera par contre récupéré) et moins rapide.
    return copieInverse ( copieInverse ( debut ) );
  }
```

6.

```
public static Maillon<Integer> enleverNegatifs ( Maillon<Integer> debut ) {
    Maillon<Integer> reponse = debut;
    // enlever tous les négatifs au début de la liste
    while ( reponse != null && reponse.info() < 0 ) {
        reponse = reponse.suivant();
    }
    // s'il reste des maillons, enlever les négatifs du reste
    if ( reponse != null ) {
        Maillon<Integer> p = reponse; // p n'est pas un maillon qui contient un négatif
        while ( p.suivant() != null ) {
            if ( p.suivant().info() < 0 ) {
                p.modifierSuivant ( p.suivant().suivant() );
            } else {
                p = p.suivant();
            }
        }

    }
    return reponse;
}
```

7.

(a) 0, 1, 2, 3, 4, 5,6, 7, 8, 9.

(b) 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23.

(c) 0, 1, 2, 3, 4, 5,6, 7, 8, 9.

(d) 0, 1, 2, 3, 4, 5,6, 7, 8, 9, 10, 11, 12.

(e) 11, 5, 8, 6, 7.

(f) 11, 17, 14, 15, 16.

8. Vous pouvez vérifier vos réponses en codant les algorithmes de tri.

9. Le tri par insertion sera le plus rapide puisqu'un seul tour de la boucle principale sera effectué. Les autres tris feront le même nombre de tours de boucles et le quicksort ne sera pas plus efficace (c'est même un cas où son efficacité sera la même que le tri par sélection et le tri bulle.)

10. La recherche séquentielle ordonnée est plus efficace lorsqu'on recherche un élément non présent puisqu'il n'est pas nécessaire de parcourir toute la liste pour conclure que l'élément est absent.

11. Dans la recherche binaire, on doit accéder en $O(1)$ (temps constant, donc indépendant de la longueur de la liste) à l'élément qui est au milieu. C'est le cas pour les tableaux et les ArrayList (structures à accès direct). Avec une liste chaînée de maillons, accéder à l'élément du milieu est proportionnel à la longueur de la liste. Cet algotithme a une piètre performance à cause de cela.

12.

(b) 25, 39, 30, 37, 35, 34.

(c) 25, 39, 100, 200, 150, 198, 160, 155.

(d) infixe : 1 5 8 9 10 12 15 17 22 25 26 27 30 32 33 34 35 36 37 39 100 107 112 120 125 130 150 160 170 180 198 200
prefixe : 25 17 10 5 1 8 9 15 12 22 39 30 27 26 37 35 34 32 33 36 100 200 150 125 120 112 107 130 198 160 180 170
postfixe : 1 9 8 5 12 15 10 22 17 26 27 33 32 34 36 35 37 30 107 112 120 130 125 170 180 160 198 150 200 100 39 25

(f) Nombre de feuilles : 10, Nombre de nœuds : 32. Hauteur de l'arbre : 8.

(g) 3.

(h) 9.

13.

```
public static int nbNoeuds ( Noeud racine ) {
    int reponse = 0;
    if ( racine != null ) {
        reponse = 1 + nbNoeuds ( racine.gauche() ) + nbNoeuds ( racine.droite() );
    }
    return reponse;
}
```

14.

```
public static int hauteur ( Noeud racine ) {
    int reponse = -1;
    if ( racine != null ) {
        reponse = 1 + Math.max ( hauteur ( racine.gauche() ), hauteur ( racine.droite() ) );
    }
    return reponse;
}
```

15.

```
public static int nbFeuilles ( Noeud racine ) {
    int reponse = 0;
    if ( racine != null ) {
        if ( racine.gauche() == null && racine.droite() == null ) {
            reponse = 1;
        }
        reponse += nbFeuilles ( racine.gauche() ) + nbFeuilles ( racine.droite() );
    }
    return reponse;
}
```

18.

```
public static int nbNoeudsSuperieursOuEgaux ( Noeud<Integer> racine, int n ) {
    // on suppose qu'il n'y a pas de doublons dans l'arbre
    int reponse = 0;
    if ( racine != null ) {
        reponse = nbNoeudsSuperieursOuEgaux ( racine.droite(), n );
                    // il faut aller à droite dans tous les cas
        int direction = -racine.info().compareTo ( n );
        if ( direction <= 0 ) {
                    // n <= racine.info() donc racine.info() est >= n
            ++reponse;
        }
        if ( direction < 0 ) {
                    // n < racine.info() donc il y a peut-être des noeuds à gauche >= n
            reponse += nbNoeudsSuperieursOuEgaux ( racine.gauche(), n );
        }
    }
    return reponse;
}
```

20.

```
public static boolean estABR ( Noeud<String> racine ) {
    boolean cEstUnABR = true;
    if ( racine != null ) {
        String info = racine.info();
        if ( racine.gauche() != null ) {
            cEstUnABR = info.compareTo ( racine.gauche().info() ) > 0;
        }
        if ( cEstUnABR && racine.droite() != null ) {
            cEstUnABR = info.compareTo ( racine.droite().info() ) < 0;
        }
        if ( cEstUnABR ) {
            cEstUnABR = estABR ( racine.gauche() ) && estABR ( racine.droite() );
        }
    }
    return cEstUnABR;
}
```

21.

```
public static Noeud copie ( Noeud racine ) {
    Noeud racineCopie = null;
    if ( racine != null ) {
        racineCopie = new Noeud ( racine.info(),
                            copie ( racine.gauche() ), copie ( racine.droite() ) );
    }
    return racineCopie;
}
```

22.

```
public static Noeud<String> construire ( ArrayList<String> parcoursPrefixe ) {
    Noeud<String> racine = null;
    if ( !parcoursPrefixe.isEmpty() ) {
        String info = parcoursPrefixe.get(0);
        // repérer les noeuds du sous-arbre gauche
        int i = 1;
        while ( i < parcoursPrefixe.size() &&
                        parcoursPrefixe.get(i).compareTo ( info ) < 0 ) {
            ++i;
        }
        ArrayList<String> gauche = new ArrayList<String>();
        gauche.addAll ( parcoursPrefixe.subList ( 1, i ) );
        ArrayList<String> droite = new ArrayList<String>();
        droite.addAll ( parcoursPrefixe.subList ( i, parcoursPrefixe.size() ) );
        racine = new Noeud<String> ( info, construire ( gauche ), construire ( droite ) );
    }
    return racine;
}
```