1. Het toepassen van abstracte datatypen:
   (a) Vereist zelfs in een programmeertaal als Java de nodige discipline
   (b) Is alleen mogelijk indien je een programmeertaal als Java gebruikt
   (c) Gaat automatisch als je de programmeertaal Java gebruikt

2. De relatie tussen *abstracte datatypen* en (*Java-*) *interfaces* is:
   (a) het zijn twee verschillende namen voor hetzelfde concept
   (b) het concept ‘interface’ is een uitbreiding van het concept ‘abstracte datatypen’
   (c) interfaces zijn een geschikte manier om abstracte datatypen in Java uit te drukken

3. Bij het bepalen van de ‘Big-Oh’ notatie voor de rekentijd (*running time*) T(n) van een algoritme kunnen we de precieze duur van een elementaire operatie negeren omdat:
   (a) we toch alleen maar een benadering willen
   (b) het asymptotisch gedrag van het algoritme er niet wezenlijk door wordt beïnvloed
   (c) de precieze duur niet goed meetbaar is

4. Welke rekentijd duidt op een *polynomiaal* algoritme?
   (a) $T(n) = O(n \log(n))$
   (b) $T(n) = O(n^2 + n^3)$
   (c) $T(n) = O(2^n)$
   (d) $T(n) = O(\log(n))$

5.Wat is de asymptotische *worst-case* rekentijd (in Big-Oh notatie) van de volgende variant van het Bubblesort algoritme:

```
void sort(String A[], int n) {
 boolean bubbled = false;
 for (int i=1; i<n; i++)  {
   bubbled = false;
   for (int j=0; j<n-i; j++) {
     if (A[j] >= A[j+1]) {
        String y = A[j]; A[j] = A[j+1]; A[j+1]= y;
        bubbled = true;
     }
   }
   if (!bubbled) break;
 }
}
```

   (a) $T(n) = O(n)$
   (b) $T(n) = O(2n + n \log(n))$
   (c) $T(n) = O(n^2)$
   (d) $T(n) = O(n \log(n))$

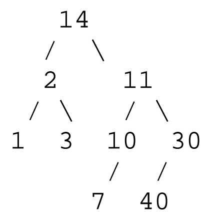
6. Hoeveel rekentijd kost het algoritme van vraag 5 in het geval dat array A[] al (oplopend) gesorteerd is?
   (a) $T(n) = O(n)$
   (b) $T(n) = O(2n + n \log(n))$
   (c) $T(n) = O(n^2)$
   (d) $T(n) = O(n \log(n))$

7. Stel, een *Stack* ADT is (efficiënt) geïmplementeerd met behulp van één Queue. Wat kun je dan over de complexiteit van Push() en Pop() zeggen?
   (a) Beide operaties zijn noodzakelijkerwijs O(n) (n is het aantal elementen op de Stack)
   (b) Een van beide kan O(1) zijn, maar de andere moet dan O(n) zijn
   (c) Een van beide kan O(n) zijn, maar de andere moet dan $O(n^2)$ zijn

8. Bezie de concepten *position en rank van een element* in een Sequence die op een Array is gebaseerd:

(a) Zowel de *rank* en de *position* van een element komen overeen met de index van het element in de Array
(b) De *rank* van een element komt overeen met de index van het element in de Array, de *position* is relatief en wordt met behulp van de index bepaald.
(c) De *position* van een element komt overeen met de index van het element in de Array, de *rank* is relatief en wordt met behulp van de index bepaald.
(d) Noch de *position* noch de *rank* hebben iets met de index van het Array van doen

9. Wat is de *in-order* volgorde van de elementen in de volgende binaire boom?



(a) 14 – 2 – 1 – 3 – 11 – 10 – 7 – 30 – 40
(b) 1 – 3 – 2 – 7 – 10 – 40 – 30 – 11 – 14
(c) 1 – 2 – 3 – 14 – 7 – 10 – 11 – 40 – 30
(d) 1 – 2 – 3 – 7 – 10 – 11 – 14 – 30 – 40

10. We willen een Euler-tour gaan gebruiken om een expressie (bijv: 3*5 / 2 + 5) te berekenen die in een binaire boom is gerepresenteerd. De Euler-tour wordt uitgevoerd met de *Template-method pattern*. In welke van de volgende vier methoden moeten we de operaties (+, -, /, *, etc.) uitvoeren?

(a) VisitLeft
(b) VisitBelow
(c) VisitRight
(d) VisitExternal

11. De *Heap-order property* houdt in dat:

(a) Het linkerkind van een knoop altijd een grotere waarde heeft dan het rechterkind
(b) Alle kinderen aan de linkerkant van een knoop kleinere waarden hebben dan de kinderen aan de rechterkant van een knoop
(c) Kinderen geen kleinere waarden hebben dan hun ouder
(d) Knopen op diepte *d* altijd grotere waarden hebben dan knopen op diepte *d*-1

12. Bij het toevoegen van een element in een Binary Search Tree

(a) Wordt dit element altijd in een externe knoop geplaatst, die daarmee intern wordt
(b) Wordt er, door verschuiven van elementen, eerst ruimte in een interne knoop gemaakt
(c) Worden er na het plaatsen in de boom één of meerdere *restructures* gedaan

13. AVL bomen en Splay-bomen maken beide gebruik van rotaties (*tri-node restructure operations*). Bij AVL bomen kan dit gebeuren bij invoegen en verwijderen, bij Splay bomen gebeurt het meermaals bij *iedere* operatie.

(a) Dit betekent dat Splay bomen altijd ongunstiger zijn dan AVL bomen omdat AVL bomen minder rotaties nodig hebben
(b) De hoogte van AVL bomen is gegarandeerd O(log n) en bij Splay bomen is dat niet zo. AVL bomen zijn dus gunstiger
(c) Splay bomen zijn gunstiger als sommige elementen vaker worden gebruikt dan andere
(d) Splay bomen zijn beter gebalanceerd dan AVL bomen en dus gunstiger

14. Voor het sorteren van kleine aantallen items:
   (a) moet je een kwadratisch algoritme gebruiken
   (b) is het onverstandig om een sorteeralgoritme met O(n log n) te gebruiken
   (c) kun je beter Insertion-Sort dan Quick-Sort gebruiken
   (d) maakt het eigenlijk niet uit welk algoritme je toepast

15. De cut-off waarde bij Quick-Sort geeft aan:
   (a) de plek waar de te sorteren lijst in tweeën wordt gedeeld
   (b) de maximale grootte van een te sorteren lijst waarop nog Quick-Sort wordt toegepast
   (c) de minimale grootte van een te sorteren deellijst waarop nog Quick-Sort wordt toegepast
   (d) de maximale diepte van de recursie

16. Dynamisch programmeren heeft de voorkeur boven Divide-and-Conquer wanneer:
   (a) er erg veel deelproblemen zijn
   (b) deelproblemen veel overlap hebben
   (c) het probleem niet in deelproblemen is op te splitsen
   (d) de oplossing van een probleem lastig uit deeloplossingen is af te leiden

17. Het voordeel van het Hirschberg algoritme voor het vinden van een *Optimal Alignment* van twee DNA-strings boven de *Longest Common Substring*-methode uit het handboek is:
   (a) Hirschberg past Divide-and-Conquer toe en geen dynamisch programmeren
   (b) De tijdcomplexiteit van het Hirschberg-algortime is veel lager
   (c) Er is (veel) minder opslagruimte nodig bij het Hirschberg-algortime
   (d) Het zijn eigenlijk identieke algoritmen: er is geen voordeel

18. Een *back edge* treedt op bij depth-first search in een gerichte graaf als:
   (a) als een knoop geen kinderen heeft
   (b) als een te bezoeken kind van een knoop al eerder is bezocht en tevens een (voor)ouder van die knoop is
   (c) als een te bezoeken kind van een knoop al eerder is bezocht maar geen (voor)ouder van die knoop is
   (d) als een pijl dwars door een eerder bezochte pijl heen loopt

19. Het aantal operaties om een substring T van string S in een *suffix trie* van S te vinden (exclusief het opbouwen van de suffix trie):
   (a) kost O(|S|) operaties
   (b) kost O(|T|) operaties
   (c) kost O(|S|+|T|) operaties
   (d) kost O(|S| x |T|) operaties

20. Bij een *region-based* k-D tree :
   (a) deelt iedere interne knoop een gebiedje in twee gelijke delen
   (b) bepaalt de verdeling van de punten in het gebiedje de manier waarop het wordt gesplitst
   (c) bepaalt de oppervlakte van een gebiedje of het wordt gesplitst

Open vraag:
Beschrijf ten minste drie mogelijke toepassingen van *Tries* in de kennistechnologie.