Aufgabe 1)

Voraussetzung für binäre Suche ist ein vorsortiertes Array, die für sequenzielle Suche nicht der Fall ist. Teilaufgabe a ist unsortiert, da 12 zwischen zwei größere Zahlen 68 und 69 steht, d. h. Für a **nur** sequenzielle Suche und für b beides Suchverfahren einsetzbar ist.

a) **Sequenzielle Suche**: Gesucht ist 12
if F [i] =12 then return i; fi

13	23	47	48	68	12	69	73	81	83	84	85	87	90	94

1. Schritt: F [i] =13!=12

13	23	47	48	68	12	69	73	81	83	84	85	87	90	94

2. Schritt: F [i] =23!=12

13	23	47	48	68	12	69	73	81	83	84	85	87	90	94

3. Schritt F [i] =47!=12

13	23	47	48	68	12	69	73	81	83	84	85	87	90	94

4. Schritt: F [i] =48!=12

13	23	47	48	68	12	69	73	81	83	84	85	87	90	94

5. Schritt: F [i] =68!=12

13	23	47	48	68	12	69	73	81	83	84	85	87	90	94

6. Schritt: F [i] =12=12, dann ist 12 gefunden und Algorithmus ist jetzt zum Ende gekommen und als Ergebnis wird (i=5) 5 geliefert.

13	23	47	48	68	12	69	73	81	83	84	85	87	90	94

b) **(I)Sequenzielle Suche**: Gesucht ist 68
if F [i] =12 then return i; fi

12	13	23	47	48	68	69	73	81	83	84	85	87	90	94

1. Schritt: F[i] =12!=68

12	13	23	47	48	68	69	73	81	83	84	85	87	90	94

2. Schritt: F[i] =13!=68

12	13	23	47	48	68	69	73	81	83	84	85	87	90	94

3. Schritt: F[i] =23!=68

12	13	23	47	48	68	69	73	81	83	84	85	87	90	94

4. Schritt: F[i] =47!=68

12	13	23	47	48	68	69	73	81	83	84	85	87	90	94

5. Schritt: F[i] =48!=68

12	13	23	47	48	68	69	73	81	83	84	85	87	90	94

6. Schritt: F[i] =68=68, dann ist 68 gefunden und Algorithmus ist jetzt zum Ende gekommen und als Ergebnis wird (i=5) 5 geliefert.

12	13	23	47	48	68	69	73	81	83	84	85	87	90	94

(II) Binäre Suche

Schritt 1: unterer Schrank ist 0, oberer Schrank ist 14, dann der Mittelwert liegt bei: m= (u+o)/2 = 7 also m=7 und vergleichen wir jetzt das Element an der Position 7 mit der gesuchten Zahl 68 (F[m] =K bzw. 73 !=68)

12	13	23	47	48	68	69	73	81	83	84	85	87	90	94

Da das Element nicht gefunden ist, soll das Intervall in zwei unterteile zerlegt werden (Teil-und hersche Prinzip). Aus dem Grund, dass 68 kleiner als 73 ist und es bei der binären Suche sich um sortiertes Array handelt, müssen wir im nächsten Schritt rekursiv die linke Seite betrachten und obere Schränke auch ändern.
If k<F[m] then return BinarySearchRec (F,K,u,m-1)

Schritt 2: unterer Schrank bleibt bei 0, oberer Schrank ist jetzt 6 und mittlerer Wert ist in Position 3 d. h. das Element 47. Da 68 großer ist als 47 müssen wie dieses Mal die rechte Seite betrachten und entsprechend den unteren Schrank ändern. Es ist genau die letzte Anweisung von else der Pseudocode für rekursiver Algorithmus return BinarySearchRec (F,K,m+1,o);

12	13	23	47	48	68	69	73	81	83	84	85	87	90	94

Schritt 3: unterer Schrank ist jetzt m+1 bzw. 4 und oberer Schrank bleibt bei 6 (wie bei Schritt 2)

12	13	23	47	48	68	69	73	81	83	84	85	87	90	94

In diesem Fall ist der mittlere Wert in der Position 5 und dazugehöriges Element ist 68. Wenn wir jetzt die Schlüssel (bei der Aufgabe gesuchte 68) mit dem mittleren Wert an der Position 5 vergleichen, merken wir, dass das aktuelle Element an dieser Position F[m]=68 gleich zum gesuchten Element K=68 ist und damit das Algorithmus erfolgreich zum Ende kommt und die Position (P=) 5 als Ergebnis zurückgeliefert wird (if k=F[m] then return m) und weitere Verkleinerungen nicht mehr notwendig sind.

Bei sequentieller Suche läuft das Algorithmus weiter bis alle Elemente durchgelaufen und verglichen werden, wenn die Liste sortiert wäre, kann man aber von weitere vergleiche verzichten (optimierung des Algorithmus).

Aufgabe 3) a)

1.Durchlauf äußeren Schleifen:

6	8	5	13	14	33	73	55	64	95

2.Durchlauf äußeren Schleifen:

6	5	8	13	14	33	55	61	73	95

3.Durchlauf äußeren Schleifen:

5	6	8	13	14	33	55	61	73	95

c) Anzahl der Vertauschungen und Vergleiche erhöht sich, da die Liste fast aufsteigend sortiert ist und weite Verschiebung kann Instabilität verursachen.

Aufgabe 4)

- Fall I: **unsortierte Daten**: Die Kompläxität bei **QuickSort** und **MergeSort** steht bei chaotischen Daten bei **O(n Log n)** im mittleren Fall, im Gegensatz zu anderen drei Verfahrens, die den Aufwand bei $O(n^2)$ liegt.

- Fall II: **aufsteigend sortierte** Daten: Die aufsteigende Sortierung ist Bestcase für **BubbleSort** und **InsertionSort** (Annahme: die Daten sollen aufsteigend sortiert werden), in diesem Fall haben die beide Sortierverfahren eine Komplexität von **O(n)**, der Aufwand bei MergeSort beträgt O(n Log n) und bei Selection- und BubbleSort bei $O(n^2)$.

- Fall III **absteigende Sortierung**: ist das Worstcase für Insertion- und BubbleSort (Annahme: die Daten sollen aufsteigend sortiert werden) und die beiden haben in diesem Fall eine Komplexität von $O(n^2)$, SelectionSort hat auch einen Aufwand von $O(n^2)$, für QuickSort ist es von Ansatz des Algorithmus und Stelle von Pivotelement abhängig. Bei der falschen Auswahl des Pivots ist die Komplexität $O(n^2)$. **MergeSort** ist hier aber mit Komplexität von **O(n Log n)** der Sieger.