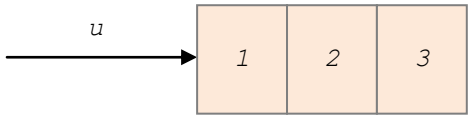
Exam: solution

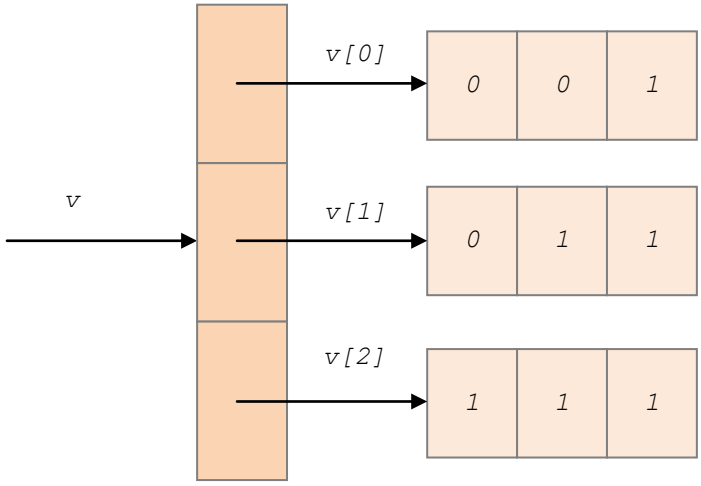
Tasks: solutions

Task 1 (3 points + 3 points)

a) (3 points)



b) (3points)



Task 2 (3 points + 3 points + 3 points)

a) (3 points)

```
public static Point nearestPoint (Point[] points, Point point)
{
    if (points.length == 0)
        throw new java.lang.IllegalArgumentException ("no points");

    Point    nearestPoint = points[0];
    double   minDistance = nearestPoint.distance (point);
    double   distance = 0;
    for (int pos = 1; pos < points.length; pos++)
    {
        distance = points[pos].distance (point);
        if (distance < minDistance)
        {
            nearestPoint = points[pos];
            minDistance = distance;
        }
    }
```

```
    return nearestPoint;
}
```

b) (3 points)

```
public static Point[] internalPoints (Point[] points, double r)
{
    // the number of points in the circle
    int    countInternalPoints = 0;
    for (Point p : points)
        if (p.getX () * p.getX () + p.getY () * p.getY () < r * r)
            countInternalPoints++;

    // the points that are in the circle
    Point[]    internalPoints = new Point[countInternalPoints];
    int    pos = 0;
    for (Point p : points)
        if (p.getX () * p.getX () + p.getY () * p.getY () < r * r)
            internalPoints[pos++] = p;

    return internalPoints;
}
```

c) (3 points)

```
Point[]    points = { new Point (3, 4),
                      new Point (1, 2),
                      new Point (5, 6),
                      new Point (4, 5) };
Point    point = new Point (1, 1);
double    radius = 7;

Point    nearestPoint = nearestPoint (points, point);
Point[]    internalPoints = internalPoints (points, radius);
```

Task 3 (3 points + 3 points + 3 points)

a) (3 points)

```
public String toString ()
{
    String    s = "";
    for (int pos = 0; pos < courseCount; pos++)
        s = s + courses[pos] + "\n";

     return s;
}
```

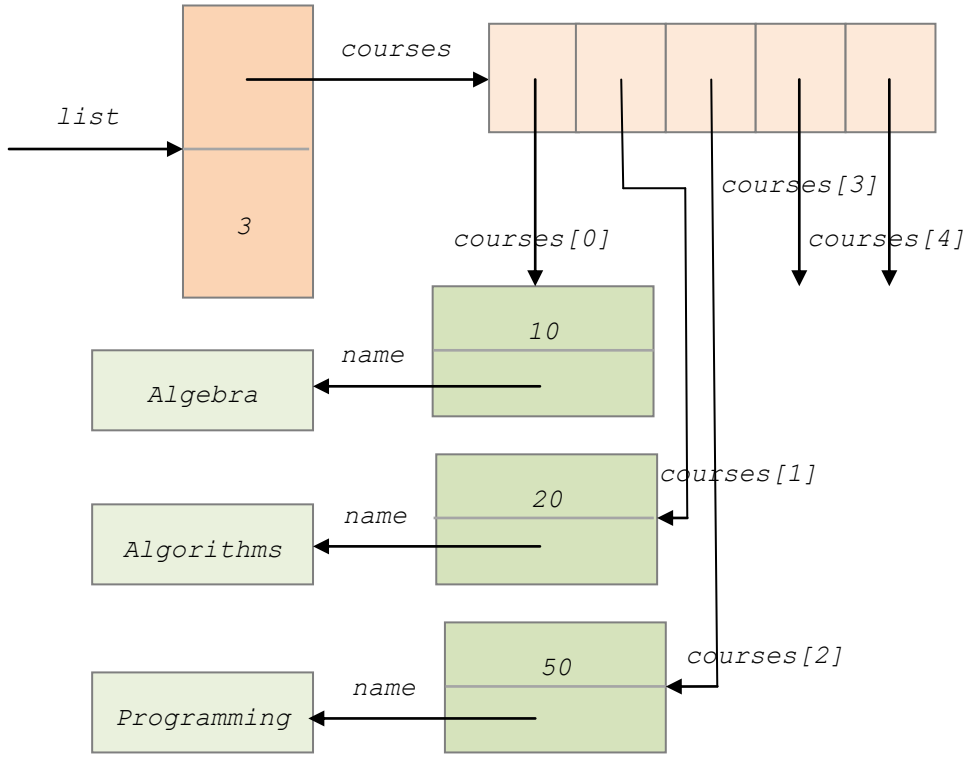
b) (3 points)

```
public void remove (Course course)
{
    int    courseIndex = -1;;
    for (int pos = 0; pos < courseCount; pos++)
        if (courses[pos].equals (course))
        {
            courseIndex = pos;
            break;
        }

    if (courseIndex != -1)
    {
        for (int pos = courseIndex; pos < courseCount - 1; pos++)
            courses[pos] = courses[pos + 1];
        courses[courseCount - 1] = null;
```

```
            courseCount--;
        }
    }
```

c) (3 points)



Task 4 (4 points + 2 points + 3 points)

a) (4 points)

```
public int sizeCompare (Rectangle rec)
{
    int    compResult = 0;
    if (this.width * this.height < rec.width * rec.height)
        compResult = -1;
    else if  (this.width * this.height > rec.width * rec.height)
        compResult = 1;

    return compResult;
}

public int sizeCompare (CharSequence seq)
{
    int    compResult = 0;
    if (this.charCount < seq.charCount)
        compResult = -1;
    else if (this.charCount > seq.charCount)
        compResult = 1;

    return compResult;
}
```

b) (2 points)

```
Rectangle    rec1 = new Rectangle (4, 3);
```

```
Rectangle    rec2 = new Rectangle (6, 5);
Rectangle    rec = Selector.oneOfTwo (rec1, rec2);
```

c) (3 points)

1+2=3

Task 5 (4 points + 5 points)

a) (4 points)

To determine the element that should be in the first position, n – 1 element comparisons are performed. To determine the element in the second position, n – 2 comparisons are required. The number of comparisons is reduced by one for each position. The total number of comparisons is:

$$(n - 1) + (n - 2) + \ldots + 1 = n(n - 1)/2$$

The corresponding complexity function is:

$$t(n) = n(n - 1)/2$$

$$t(n) = n^2/2 - n/2$$

For large n the term n^2 dominates. Therefore:

$$t(n) \in \theta(n^2)$$

The algorithm is quadratic in terms of element comparisons.

You put n – 1 elements in the correct place, and thereby all elements are sorted. When an element is put in the correct place, it swaps positions with the element already there. If an element is already in the correct position, it does not need to swap positions with any other element. This means that in the worst case, there are n – 1 element exchanges.

The worst case time complexity of the algorithm, in terms of the number of element exchanges, can be given by the following complexity function:

$$w(n) = n - 1$$

$$w(n) \in \Theta(n)$$

The algorithm is linear in terms of element exchanges, in the worst case.

c) (5 points)

```
public static void sort (String[] elements)
{
    int    lastPos = elements.length - 1;
    int    minPos = 0;
    String    e = "";
    for (int pos = 0; pos < lastPos; pos++)
    {
        minPos = pos;
        for (int p = pos + 1; p <= lastPos; p++)
            if (elements[p].compareTo (elements[minPos]) < 0)
                minPos = p;

        if (minPos != pos)
        {
            e = elements[pos];
            elements[pos] = elements[minPos];
            elements[minPos] = e;
        }
    }
}
```