Exercise Session 9

Exercise 1

A

```
Memoized-Fib(n)
    let ns be a new array of size n
    for i = 1 to n
        ns[i] = null
    fib(n, ns)

fib(n, ns)
    if ns[n] != null
        return ns[n]
    elif n < 2
        ns[n] = 1
        return 1
    else
        ns[n] = Fib-Rec(n - 1) + Fib-Rec(n - 2)
        return ns[n]
```

B

Using dynamic programming we have made so that every sub problem is only calculated once. Thus the running time should be $\Theta(n)$. We can see this by the fact that we only calculate the previous steps fibonacci numbers if we do not know the current one. Once we know the current fibo number, then we save it, thus every number is only calculated once.

C

When we first call Memoized-Fib we initialize an array of size n. Also when we call fib we will keep the recurrence going until we reach a base case, thus we will also have a linear call stack. So i do not know if this should be $\Theta(n)$ or $\Theta(n^2)$.

Exercise 2

A

Because for every number in the list we would have to check every number after that as well, to see it was bigger. That means that when you add another element, you have to check every sub-sequence, containing that number and every sub-sequence that does not, every time you add a number you are effectively doubling the amount of possible sub-sequences.

B

```
Sub-Seq(A, i)
    if i == 1
        return 1

    let j = i - 1
    while A[i] < A[j] and j != 0
        j--

    if j == 0
        return 1
    else
        return Sub-Seq(A, j) - 1
```

C

```
BottomUp-LNDS(A)
    let vs = []
        vs[1] = 1
        ss = []
        ss[1] = A[1]

    for i = 2 to A.len
        let j = i - 1
        while A[i] < A[j] and j != 0
            j--

        if j != 0
            vs[i] = vs[j] + 1
            ss[i] = ss[j] ++ A[i]
        else
            ls[i] = 1
            s[i] = A[i]

    let v = max(vs)

    return v and s[ vs.indexOf(val) ]
```

D

```
Print-LNDS(A)
    let [v, s] = BottomUp-LNDS(A)
    print s
```

Exercise 3

A

This has got to be wrong

```
Knapsack(vs, ws, W)
  let tvs = []
    ts  = []
    tvs[0, 0] = 0
    j = W
  for i = 1 to vs.len
    if ws[i] > j
      tvs[i, j] = tvs[i - 1, j]
      is[i, j] = is[i - 1, j]
    else
      tvs[i, j] =
        max(vs[i] + tvs[i - 1, j - ws[i], tvs[i - 1, j])
      is[i, j] = is[i - 1, j - ws[i]] ++ i

  let v = max(tvs)
    s = m.indexOf(v)

  return v and s
```

B

```
Print-Knapsoack(vs, ws, W)
    let [v, s] = Knapsack(ws, ws, W)
    print s
```

Exercise 4

```
lego(l, ss)
    let vs = []
    for i = 1 to l
        vs[i] = POS_INF
        v = POS_INF
        for s in ss
            if i - s > 0
                v = min(v, 1 + vs[i - s])
            if i - s == 0
                v = 0
                break
        vs[i] = v
```

The worst case running time should be $O(max(n^2, m^2))$ where n and m are the length of `l`and `ss`.