1. Prove $n^2 \in \mathcal{O}(n^3)$:
   $\lim\limits_{n\to\infty} \dfrac{n^2}{n^3} = 0$
   $0 < \infty$ therefore by limit definition of asymptotic relations $n^2 \in \mathcal{O}(n^3)$

   Prove $n^3 \notin \mathcal{O}(n^2)$:
   $\lim\limits_{n\to\infty} \dfrac{n^3}{n^2} = \infty$
   $\infty \not< \infty$ therefore by limit definition of asymptotic relations $n^3 \notin \mathcal{O}(n^2)$

2. Find the time complexities of the following algorithms

   (a) LinearSearch-1
   Undefined. Program never terminates for $n > 0$ because i never increments.

   (b) LinearSearch-2
   Undefined. First iteration of while loop accesses A at 0, but should be 1-based index of $A[1]$ and should therefore run while $i \leq n$.

   (c) Factorial
   Undefined. No base case for $n = 1$.

3.

   (i)

```
FindRepeatedNumberNaive(A[1...n])
for i ← 1 to n do
    for j ← i to n do
        if A[i] = A[j] then
            return A[i]
return −1
```

   (ii)

```
FindRepeatedNumberEfficient(A[1...n])
Create an array found[1...(n − 1)]
for i ← 1 to n do
    value ← A[i]
    if found[value] = true then
        return value
    found[value] ← true
return −1
```

4.

(i)

```
GroupingNaive(A[1...n])
remove_index ← 1
for i ← 1 to n do
    element ← A[i]
    if IsPerfectSquare(element) then
        remove_index ← i
        for j ← remove − 1 down to 0 do
            A[j + 1] ← A[j]
        A[1] ← element
```

(ii)

```
GroupingBetter(A[1...n])
Create an array left[1...n]
Create an array right[1...n]
left_count ← 1; right_count ← 1
for i ← 1 to n do
    num ← A[i]
    if IsPerfectSquare(num) then
        left[left_count] ← num
        left_count ← left_count + 1
    else
        right[right_count] ← num
        right_count ← right_count + 1
for i ← 1 to left_count do
    A[i] ← left[i]
for i ← 1 to right_count do
    A[left_count + i] ← right[i]
```

```
GroupingBest(A[1...n])
left ← 1; right ← n
while left < right do
    a ← A[left]; b ← A[right]
    asq ← IsPerfectSquare(a); bsq ← IsPerfectSquare(b)
    if not asq and bsq then
        Swap(a, b)
    if asq then
        left ← left + 1
    if not bsq then
        right ← right − 1
```

5.

(i)

```
JosephusProblemArray(n, k, j)
Create an array people[1...n] ← [1...n]
capped ← (n % k = 0) ? j : Min(j, n)
index ← 0
visited_since_last_kill ← k
killed ← 0
last_killed ← 0
while killed < capped do
    i ← 1 + index % n
    person ← people[i]
    if person = 0 then
        index ← index + 1
    else if visited_since_last_kill = k then
        last_killed ← person
        people[i] ← 0
        killed ← killed + 1
        visited_since_last_kill ← 0
    else
        visited_since_last_kill ← visited_since_last_kill + 1
    index ← index + 1
return last_killed
```

(ii)

```
JosephusProblemCLL(n, k, j)
Create a CircularSinglyLinkedList people
for i ← 1 to n do
    people.AddLast(i)
max_iterations ← (n % k = 0) ? ⌊n / k⌋ : n
iterations ← Min(j, max_iterations)
killed ← 0
visited ← 0
final_kill ← 0
while killed < iterations do
    person ← people.First()
    people.RemoveFirst()
    if visited % k = 0 then
        killed ← killed + 1
        final_kill ← person
    else
        people.AddLast(person)
    visited ← visited + 1
return final_kill
```

6.

(i)

```
MaximizeProductNaive(A[1...n])
max ← A[1] × A[2]
for i ← 1 to n do
    for j ← i + 1 to n do
        max ← Max(max, A[i] × A[j])
return max
```

(ii)

```
MaximizeProductBetter(A[1...n])
Sort(A[1...n])
neg_max ← A[1] × A[2]; pos_max ← A[n] × A[n − 1]
return Max(neg_max, pos_max)
```

(iii)

```
MaxProductBest(A[1...n])
neg_max ← A[1]; neg_sec_max ← A[1]
pos_max ← A[1]; pos_sec_max ← A[1]
for i ← 1 to n do
    num ← A[i]
    if num ≤ neg_max then
        neg_sec_max ← neg_max
        neg_max ← num
    if num ≥ pos_max then
        pos_sec_max ← pos_max
        pos_max ← num
return max(neg_max × neg_sec_max, pos_max × pos_sec_max)
```