1. Insertion sort

```python
def insertSort(a):
    
    /*
    BEHAVIOUR: Run the insertsort algorithm on the integer array a, sorting
    it in place.
    PRECONDITION: array a contains len(a) integer values.
    POSTCONDITION: array a contains the same integer values as before, but
    now they are sorted in ascending order.
    */

    for i from 1 included to len(a) excluded:

        /*
        ASSERT: the first i positions are already sorted.
        Insert a[i] where it belongs within a[0:i].
        */

        j=i-1
        while j >= 0 and a[j] > a[j + 1]:
            swap(a[j], a[j + 1])
            j=j-1

(a) Assume that each \texttt{swap}(x, y) means three assignments (namely \texttt{tmp} = x; x = y; y = \texttt{tmp}). Improve the insertsort algorithm pseudocode shown in the handout to reduce the number of assignments performed in the inner loop.

(b) Provide a useful invariant for the inner loop of insertion sort, in the form of an assertion to be inserted between the “while” line and the “swap” line.

(c) What is the asymptotic complexity of the variant of insertsort that does fewer swaps?

2. Selection sort

```python
def selectSort(a):
    
    /*
    BEHAVIOUR: Run the selectsort algorithm on the integer array a, sorting
    it in place.
    PRECONDITION: array a contains len(a) integer values.
    POSTCONDITION: array a contains the same integer values as before, but
    now they are sorted in ascending order.
    */
for k from 0 included to len(a) excluded:

  /*
  ASSERT: the array positions before a[k] are already sorted.
  Find the smallest element in a[k:END] and swap it into a[k].
  */

  iMin = k
  for j from iMin + 1 included to len(a) excluded:
    if a[j] < a[iMin]:
      iMin = j
  swap(a[k], a[iMin])

When looking for the minimum of \( m \) items, every time one of the \( m - 1 \) comparisons fails the best-so-far minimum must be updated. Give a permutation of the numbers from 1 to 7 that, if fed to the *Selection sort* algorithm, maximizes the number of times that the above mentioned comparison fails.

3. **Bubble sort**

```python
def bubbleSort(a):

  /*
  BEHAVIOUR: Run the bubble sort algorithm on the integer array a, sorting it in place.
  PRECONDITION: array a contains len(a) integer values.
  POSTCONDITION: array a contains the same integer values as before, but now they are sorted in ascending order.
  */

  repeat:
    // Go through all the elements once, swapping any that are out of order
    didSomeSwapsInThisPass = False
    for k from 0 included to len(a) - 1 excluded:
      if a[k] > a[k + 1]:
        swap(a[k], a[k + 1])
        didSomeSwapsInThisPass = True
    until didSomeSwapsInThisPass == False

Prove that Bubble sort will never have to perform more than \( n \) passes of the outer loop.
4. Mergesort

```python
def mergeSort(a):
    
    /*
    *** DISCLAIMER: this is purposefully NOT a model of good code
    (indeed it may hide subtle bugs) but it is a useful starting
    point for our discussion. ***
    
    BEHAVIOUR: Run the merge sort algorithm on the integer array a, 
    returning a sorted version of the array as the result. (Note that 
    the array is NOT sorted in place.)
    PRECONDITION: array a contains len(a) integer values.
    POSTCONDITION: a new array is returned that contains the same 
    integer values originally in a, but sorted in ascending order. 
    */
    
    if len(a) < 2:
        // ASSERT: a is already sorted, so return it as is
        return a
    
    // Split array a into two smaller arrays a1 and a2 and sort these 
    recursively
    h = int(len(a) / 2)
    a1 = mergeSort(a[0:h])
    a2 = mergeSort(a[h:END])

    // Form a new array a3 by merging a1 and a2
    a3 = new empty array of size len(a)
    i1=0  // indexintoa1
    i2=0  // indexintoa2
    i3=0  // indexintoa3
    while i1 < len(a1) or i2 < len(a2)
        // ASSERT: i3 < len(a3)
        a3[i3] = smallest(a1, i1, a2, i2) // updates i1 or i2 too
        i3 = i3 + 1
        // ASSERT: i3 == len(a3)
    
    return a3
```

(a) Can you spot any problems with the suggestion of replacing the line that assigns to `a3[i3]` with the more explicit and obvious `a3[i3] = min(a1[i1], a2[i2])`? What would be your preferred way of solving such problems? If you prefer to leave that line as it is, how would you implement the procedure `smallest` it calls? What are the trade-offs between your chosen method and any alternatives?
(b) How can you do mergesort in \(n/2\) space?

5. Quicksort [Refer pg 28 of the notes]
   Can picking the pivot at random really make any difference to the expected performance? How will it affect the average case? The worst case? Discuss.

6. Comparisons
   (a) What is the smallest number of pairwise comparisons you need to perform to find the smallest of \(n\) items?
   (b) And to find the second smallest?