Tutorial 5 solutions

1. (a) The algorithm described here will find if such a position exists between two positions `first` and `last` of the array, including the endpoints. The idea is simple: we look at the middle position, and then recurse on the either the first half or the second half of the array depending on the result of the comparison.

```
Algorithm findPosition(A, first, last)

if first = last then
    if A[first] = first then
        return first
    endif
    return false
endif

mid ← ⌊ (first + last)/2 ⌋
if A[mid] = mid then
    return mid
endif

if A[mid] < mid then
    return findPosition(A, mid+1, last)
else
    return findPosition(A, first, mid-1)
endif
```

   (b) Here is a complete proof of correctness for the algorithm, using mathematical induction on n, where n is the number of elements of `A` from position `first` to position `last`. Clearly the algorithm will return the correct answer if $n = 1$, since this is the case where `first = last`.

   So suppose that `first` $<$ `last`. If `A[mid] = mid` then the algorithm will return the correct position. Consider now the case `A[mid]` $<$ `mid`.

   Claim: For every non-negative integer $j \leq$ `mid`, `A[mid - j]` $<$ `mid - j`.

   Proof of claim: By induction on j. When $j = 0$, we are comparing `A[mid]` to `mid`, which is true since this is the case we are examining. Suppose now that the claim holds for j. Because `A` contains elements that are distinct and sorted, `A[mid - (j + 1)]` $\leq$ `A[mid - j] - 1` $<$ `(mid - j) - 1 = mid - (j + 1)`. QED

   Thus, a position i such that `A[i] = i` can not satisfy `i` $\leq$ `mid`, and hence the solution can only be found in between positions `mid + 1` and `last`.

   Finally, we need to consider the case where `A[mid]` $>$ `mid`. The proof of this case is symmetric to the previous one, and completes the induction step and the proof of correctness of the algorithm.

(c) The running time of the algorithm satisfies the recurrence relation $T(n) \leq T(\lfloor n/2 \rfloor) + \Theta(1)$ with $T(1) \in \Theta(1)$ and so by Case 2 of the Master theorem, $T(n) \in \Theta(logn)$.

2. (a) We mimic the `Mergesort` algorithm. The algorithm will return the 4-tuples sorted from left to right (the ordering simplifies the merge operation considerably).

```
Algorithm OneDSurfaceRemoval(S, first, last)

if first = last then
    return S[first ... first]
endif

mid ← ⌊ (first + last)/2 ⌋
scene1 ← OneDSurfaceRemoval(S, first, mid)
scene2 ← OneDSurfaceRemoval(S, mid + 1, last)
return OneDSurfaceMerge(scene1, scene2)
```

The merge stage takes in two arrays of intervals (with depths and colors) that are already sorted from left to right. No two intervals in one array overlap, although of course intervals in the first array may overlap with intervals in the second array. The algorithm proceeds from left to right. It maintains a variable `xmin` that contains the smallest value of x that has not yet been assigned a color. At each step, we consider one interval from each subproblem, and decide how to deal with them up to the end of the first one to terminate. There are 3 cases:

i. There is no interval immediately to the right of $x =$ `xmin`. In this case we simply advance `xmin` to the beginning of the next interval. No output is produced.

ii. There is only one interval immediately to the right of $x =$ `xmin`. We add the portion of this interval that comes before the start of the first interval in the other array to our output, and update `xmin`.

iii. There are two intervals immediately to the right of $x =$ `xmin`. Only one of them will be visible until one of the two terminates. We add that one to the output, and then update `xmin`.

In every case we move `xmin` from one endpoint of an interval to another endpoint of an interval. Since there are at most $O(n)$ interval endpoints when we have n intervals, this merge process thus runs in $O(n)$ time. Pseudo-code follows, although we didn't expect you to come up with this during the tutorials. Note that this pseudo-code may produce intervals with 0 length in some cases. It may also break a single interval into several smaller intervals with the same depth and color. A single pass through the list it produces is sufficient to clean up those artifacts (we haven't included pseudo-code for that),.

```
Algorithm OneDSurfaceMerge(scene1, scene2)

i ← 0
j ← 0
S ← { }
xmin ← −∞

while (i < length[scene1] and j < length[scene2]) do
  //
  // Case 1
  //
  if xmin < scene1[i].xl and xmin < scene2[j].xl then
    xmin ← min (scene1[i].xl, scene2[j].xl)

  //
  // Case 2 (with one interval).
  //
  else if xmin ≥ scene1[i].xl and xmin < scene2[j].xl then
    if scene1[i].xr ≤ scene2[j].xl then
      add (xmin, scene1[i].xr, scene1[i].depth, scene1[i].col) to S
      xmin ← scene1[i].xr
      i ← i + 1
    else
      add (xmin, scene2[j].xl, scene1[i].depth, scene1[i].col) to S
      xmin ← scene2[j].xl
    endif

  //
  // Case 2 again (with the other interval).
  //
  else if xmin ≥ scene2[j].xl and xmin < scene1[i].xl then
    if scene2[j].xr ≤ scene1[i].xl then
      add (xmin, scene2[j].xr, scene2[j].depth, scene2[j].col) to S
      xmin ← scene2[j].xr
      j ← j + 1
    else
      add (xmin, scene1[i].xl, scene2[j].depth, scene2[j].col) to S
      xmin ← scene1[i].xl
    endif

  //
  // Case 3 (the interval in scene 1 finishes first, or maybe both
  // end at the same place).
  //
  else if scene1[i].xr ≤ scene2[j].xr
    if scene1[i].depth < scene2[j].depth then
      add (xmin, scene1[i].xr, scene1[i].depth, scene1[i].col) to S
    else
      add (xmin, scene1[i].xr, scene2[j].depth, scene2[j].col) to S
    endif
    xmin ← scene1[i].xr
    i ← i + 1

  //
  // Case 3 again (the interval in scene 2 finishes first).
  //
  else
    if scene1[i].depth < scene2[j].depth then
      add (xmin, scene2[j].xr, scene1[i].depth, scene1[i].col) to S
    else
      add (xmin, scene2[j].xr, scene2[j].depth, scene2[j].col) to S
    endif
    xmin ← scene2[j].xr
    j ← j + 1
  endif
endwhile

//
// Now we need to collect the remaining intervals in each scene. Only
// one of these loops will actually execute.
//
while i < length[scene1] do
    add (xmin, scene1[i].xr, scene1[i].depth, scene1[i].col) to S
    i ← i + 1
    if i < length[scene1] then
      xmin ← scene1[i].xl
    endif
endwhile

while j < length[scene2] do
    add (xmin, scene2[j].xr, scene2[j].depth, scene2[j].col) to S
    j ← j + 1
    if j < length[scene2] then
      xmin ← scene2[j].xl
    endif
endwhile

return S
```

(b) The running time of the algorithm can be described by the same recurrence relation as the running time of `Mergesort`, and so it is in $\Theta(n \log n)$.