Homework 1

Tripp Lamb

2026 Jan 22

CS 617-01

Problem 1

Page 41, Problem 2.4 Inversion numbers. you only need to present the pseudo-code part. Please refer to https://www.cs.mcgill.ca/~akroit/math/compsci/Cormen%20Introduction%20to%20Algorithms.pdf

> *Let A[1..n] be an array of n distinct numbers. If i < j and A[i] > A[j], then the pair (i, j) is called an inversion of A.*
>
> *Give an algorithm that determines the number of inversions in any permutation on n elements in Θ(n lg n) worst-case time. (Hint: Modify merge sort.)*

```
// `A` is an array of `n` numbers
// `p` is the start index
// `r` is the end index
// `count` is the local running count of the number of available
inversions

top-level call: determine_number_of_inversions(A, 1, n)

determine_inversions(A, p, r) -> count

    count = 0

    if p < r

      q = floor((p + r)/2)
```

```
      count += determine_number_of_inversions(A, p, q)
      count += determine_number_of_inversions(A, q+1, r)

      count += merge(A, p, q, r)

    return count


// Do normal merge sort, but every time the right side of the
//   sub array is picked increment the counter by the number
//   of elements remaining to be sorted in the left side

merge(A, p, q, r) -> count
  n_l = q - p + 1
  n_r = r - q
  let L[1:n_l], and R[1:n_r] be new arrays

  for(i=1; i<=n_l; i++)
    L[i] = A[p+i-1]
  for(i=1; i<=n_r; i++)
    R[i] = A[q+i]

  i = 1
  j = 1
  k = p
  count = 0

  while i <= n_l and j <= n_r
    if L[i] <= R[j]
      A[k] = L[i]
      i += 1
    else
      count += (n_l - i + 1)
      A[k] = R[j]
      j += 1

    k += 1

  while i <= n_l
    A[k] = L[i]
```

```
    i += 1
    k += 1

  while j <= n_r
    A[k] = R[j]
    j += 1
    k += 1

  return count
```

Problem 2

Present a recursive pseudo-code for insertion sort

(Iterative version for reference)

```
insertion_sort_iter(A)
  // `A` is an index-1 based array of numbers
  // `n` is len(A)

  for(ki=2; ki<=n; ki++) //key index
    key = A[ki]
    si = ki //sort index
    while si > 1 && key < A[si-1]
        A[si] = A[si-1]
        si -= 1
    A[si] = key
```

Recursive version:

```
// `A` is an index-1 based array of numbers
// `n` is len(A)
top-level call: insertion_sort_rec(A, n)

insertion_sort_rec(A, ei)

  if ei > 1
    key = A[ei]
    insertion_sort_rec(A, ei-1)
```

```
    i = ei

    while i > 1 && key < A[i-1]
      A[i] = A[i-1]
      i -= 1

    A[i] = key
```

Problem 3

Consider the Hanoi's Tower pseudocode presented in class, write a piece of pseudocode to print the k-th step when moving n plates from the Peg Origin to the Peg Destination with help from the Peg Auxiliary. Here, 1<=k<=2^n-1.

This is not the most efficient way as it continues to solve the entire Hanoi problem, but it is the simplest change to the code in class to give the answer.

```
// according to the psuedo code language this is pass-by-value the
// algorithm returns the incremented step number, changing `curr`
// internally does not change the passed value

// `n` is the number of the current plate, larger number = bigger
plate
// `from` is the peg number the current plate is moving from
// `to` is the peg number the current plate is moving to
// `hold` is the peg number not part of this move
// `curr` is the step number of the current move
// `k` is the constant number, at which step to output

top-level call: hanoi_solve(n, 1, 3, 2, 0, k)

hanoi_solve(n, from, to, hold, curr, k) -> next

  if(n > 1)
    curr = hanoi_solve(n-1, from, hold, to, curr, k)

  curr += 1
```

```
  if (curr == k)
    print from, '->', to

  if(n > 1)
    curr = hanoi_solve(n-1, hold, to, from, curr, k)

  return curr
```

Problem 4

Consider a strictly convex array A where the numbers in A first strictly increase, then the numbers strictly decrease. Write a recursive and iterative pseudo-code to find the unique largest number in A.

```
// `A` is 1-indexed distinct convex array of length `n`.

convex_find_max_iter(A)

  left = 1
  right = n

  while left < right
    i = floor((left + right)/2)

    if (A[i] < A[i+1])
      left = i + 1
    else
      right = i

  return A[left]
```

```
// `A` is 1-indexed distinct convex array of length `n`.

top-level call: convex_find_max_recursive(A, 1, n)

convex_find_max_recursive(A, left, right)

  i = floor((left + right)/2)
```

```
if (left == right)
  return A[left]
else if (A[i] < A[i+1])
  return convex_find_max_recursive(A, i+1, right)
else
  return convex_find_max_recursive(A, left, i)
```