Assignment 10

C-4.16 Given a sequence S of n comparable elements, describe an efficient method for determining whether there are two equal elements in S. What is the running time of your method?

```
hasDuplicate(S):
    sort(S)                  // O(n log n)
    for i from 1 to n-1:     // O(n)
        if S[i] == S[i-1]:
            return True
    return False
```

Running Time

Sorting: O(nlogn)

Scan: O(n)

O(nlogn) time

C-4-19 Let S be a sequence of n elements on which a total order relation is defined. An inversion in S is a pair of elements x and y such that x appears before y in S but $x > y$. Describe an algorithm running in $O(n \log n)$ time for determining the number of inversions in S, i.e., the number of inversions of element x in S is the count of the number of elements that came before x in the original input but are greater than x and should be after x in the sorted ordering. Hint: modify the merge-sort algorithm to solve this problem.

```
countInversions(S):
    if length(S) ≤ 1:
        return (S, 0)
    mid = length(S)//2
    (L, leftInv) = countInversions(S[0:mid])
    (R, rightInv) = countInversions(S[mid:end])
    (merged, splitInv) = mergeAndCount(L, R)
    return (merged, leftInv + rightInv + splitInv)

mergeAndCount(L, R):
    i = j = invCount = 0
    merged = []
    while i < len(L) and j < len(R):
        if L[i] ≤ R[j]:
            merged.append(L[i])
            i++
        else:
            merged.append(R[j])
            invCount += (len(L) - i)
            j++
    append remaining L and R to merged
    return (merged, invCount)
```

Old exam questions:
A. Given a Tree T, write a pseudo code algorithm findDeepestNodes(T), that returns a Sequence of pairs (v, d) where v is an internal node of tree T and d is the depth of v in T. The function must return all internal nodes that are at the maximum depth (no other nodes). What is the time complexity of your algorithm?

```
findDeepestNodes(T):
    maxDepth = -1
    result = []

    def dfs(v, depth):
        if isInternal(v):
            if depth > maxDepth:
                maxDepth = depth
                result.clear()
                result.append((v, depth))
            elif depth == maxDepth:
                result.append((v, depth))
        for each child c of v:
            dfs(c, depth + 1)

    dfs(T.root, 0)
    return result
```

B. isExclusiveOr(A, B, C)

```
isExclusiveOr(A, B, C):
    E = buildExclusive(A, B)
    return matchRecursive(C, E)

buildExclusive(A, B):
    if A.isEmpty() and B.isEmpty():
        return emptyList
    if not A.isEmpty():
        a = A.first()
        rest = buildExclusive(A.after(a), B)
        if not B.contains(a):
            rest.add(a)
        return rest
    if not B.isEmpty():
        b = B.first()
        rest = buildExclusive(A, B.after(b))
        if not A.contains(b):
            rest.add(b)
        return rest

matchRecursive(C, E):
    if C.isEmpty() and E.isEmpty():
        return True
    if C.isEmpty() or E.isEmpty():
        return False
    c = C.first()
    if E.contains(c):
        E.remove(c)
        return matchRecursive(C.after(c), E)
    else:
        return False
```

Time Complexity

Building exclusive set: O((|A|+|B|)2) (because of contains calls).

Matching: O(|C|*|E|).

Can be improved to O(n) with hashing

C. Sorting Thousands of Paper Documents

Plan

Distribute across 12 tables:

Break into 12 roughly equal piles.

Sort each table independently (human quicksort or by last name initial).

Merge tables by combining in alphabetical order.

This mimics external merge sort for large data.

D. createBST(S) from Sorted Sequence

```
createBST(S):
    T = emptyTree()
    def build(T, S):
        if S.isEmpty():
            return
        mid = len(S)//2
        v = insertRoot(S[mid]) if T.isEmpty() else insertLeft/Right appropriately
        build(T, S[0:mid])
        build(T, S[mid+1:end])
    build(T, S)
    return T
```

Time Complexity

Each element inserted once: O(n).

C-4.25: Nuts and Bolts Matching

```
matchNutsBolts(nuts, bolts):
    if nuts.length ≤ 1:
        return
    pivotNut = nuts[0]
    pivotBolt = partition(bolts, pivotNut)
    partition(nuts, pivotBolt)
    leftSize = index of pivotNut
    matchNutsBolts(nuts[0:leftSize], bolts[0:leftSize])
    matchNutsBolts(nuts[leftSize+1:end], bolts[leftSize+1:end])
```

Running Time

Like QuickSort: Average O(nlogn) comparisons.