Lab 3-1 B: Divide and Conquer Problems

Goal: Practice in applying divide and conquer to problem solving

1. **Entry = Index:** Suppose that you are given a sorted list of distinct integers $\{a_1; a_2; : : : a_n\}$. Give a divide-and conquer algorithm that determines whether there exists an index i such that $a_i = i$. For example, in { -10;-4; 3; 41}, $a_3 = 3$, but in {4; 7; 19; 20} there is no such i. State the recurrence relation for the running time and the running time.

Algorithm:

```
EntryIdx(S[], left, right)
  If right >= 1 then
      Mid = 1 + (right - 1) / 2
      If S[mid] == mid then
          Return mid
      If S[mid] > mid
          Return EntryIdx(S[], left, mid - 1)
      Return EntryIdx(S[], mid + 1, right)
  Return -1
```

Analysis:

Recurrence Relation: T(n) = T(n / 2) + O(1)
Asymptotic Running Time: O(log n)

2. Let M be an n x n matrix of integers in which the integers in every row are in increasing order from smallest row index to largest row index and the integers in every column is in increasing order from smallest column index to largest column index.
   Design and efficient algorithm (e.g. O(n)) that finds a given integer or concludes that the integer is not in the array.

Example

```
int[][] board1 = new int[][] {
{1, 2, 8, 9},
{3, 6, 12, 13},
{7,10,13, 29},
{10, 11, 28, 30}
};
int target = 12;
```

Algorithm:

```
SearchMatrix(S[][], i, j, x) //first call i = 0 and j = n -1
  If i < n and j >= 0
      If S[i][j] == x
          Return true
      If S[i][j] > x
          Return SearchMatrix(S[][], i, j-1, x)
      Return SearchMatrix(S[][], i + 1, j, x)
```

Return false

Analysis:

Recurrence Relation: T(n) = T(n*n / n) + O(1)
Asymptotic Running Time: O(n)

3. **Stock Price Analysis**: You are working for a small stock investment company that wants to look for patterns in optimal trading days in a given time period of n days. They want to find the best **pair** of days in a period of n days to buy a stock on the first day of the pair and sell it on the second day of the pair. That is, they want the biggest positive difference between the selling price on the second day and the buying price on the first day. Assume for simplicity that the buying and selling price on a given day are the same. Assume you know the stock price for every day.
   Specify an Θ (n log n) **Divide and Conquer algorithm.** (Note: there are Θ (n) solutions but that is not what is asked for here)

Algorithm:

```
StockAnalysis(A[], low, high)
    If low == high
        Return (start, start)
    Mid = (low + high) / 2
    Left = StockAnalysis(A[], low, mid)
    Right= StockAnalysis(A[], mid + 1, high)
    min = low
    For i from low to mid
        If A[i] < A[min] then
            min = i
    max = mid
    For i from mid to high do
        If A[i] > A[max] then
            Max = i
    Return max(left, right, (min, max)) //return max diff of pairs
```

Analysis:

Recurrence Relation: T(n) = 2T(n / 2) + O(n)
Asymptotic Running Time: n log n

Lab 3-2 B: Divide and Conquer Problems

Goal: Review problems for Quick Sort and Quick Select

Submit rigorous solutions to these problems

4. **Minimizing Weighing (warm up):** Suppose you are given $n = 3^k$ marbles that look identical, with one special marble that weighs more than the other marbles. You are also given a balancing scale that takes two items (or sets of items) and compares their weights. Design and analyze a divide and conquer algorithm to

find the heavy marble using the balancing scale at most k times. Give the recurrence relation for the running time. Apply the Master Theorem to show that the running time is k.

Algorithm:

```
FindMarble(S[0, n - 1])
Weight = compare(S[0, n/2 - 1], S[n/2 + 1, n - 1] ) //return -1 if first half heavier, 0 if equal, and 1 if
                                                    second half heavier
If (weight == 0) then
        Return S[n/2]
If (weight == -1) then
        Return FindMarble(S[0, n/2])
Else do
        Return FindMarble(S[n/2, n - 1])
```

Analysis:

Recurrence Relation:_ $T(n) = T(n / 2) + O(1)$

Values of: a: 1 b: 2 d: 0

Asymptotic Running Time: log n

5. **Maximum Sum Contiguous Subsequence**: In computer science, the maximum subarray problem is the task of finding the contiguous subarray within a one-dimensional array of numbers which has the largest sum. For example, for the sequence of values −2, 1, −3, 4, −1, 2, 1, −5, 4; the contiguous subarray with the largest sum is 4, −1, 2, 1, with sum 6.

Algorithm:

```
MaxSumArray(arr[], left, right)
        If left == right then
                Return arr[1]
        Mid = left + right / 2
        Return max(MaxSumArray(arr[], left, mid), MaxSumArray(arr[], mid + 1, right),
                        MaxCross(arr[], left, mid, right))

MaxCross(arr[], left, mid, right)
        Sum = 0;
        Lsum = arr[left]
        For i from left + 1 to mid do
                Sum += arr[i]
                If  sum > lsum
                        Lsum = sum;
        Sum = 0
        Rsum = arr[mid + 1]
```

```
For i from mid + 2 to right do
        Sum += arr[i]
        If sum > rsum then
                Rsum = sum
Return max(lsum, rsum, lsum + rsum)
```

Analysis:

Recurrence Relation: T(n) = 2T(n / 2) + O(n)

Values of: a: 2 b: 2 d: 1

Asymptotic Running Time: n log n

6. **k-way merge revisited**: In the lab on Heap Sort, you found a way using a heap to merge k sorted lists each with n/k items each into a single sorted list of n items of O(n log k) complexity. In this lab, you goal is to find a divide and conquer algorithm that is also more efficient than the brute force approach. The brute force approach is conceptually to:
   - merge two of the lists into a list – call it result2
   - merge a third list with result – call it result3
   - …
   - until you get result n (obviously you would most likely need to be careful about storage constrains but ignore that here

   a. Show the time complexity of this brute force algorithm Θ(kn)

   T(n) = kO(n), since you are merging lists, each merge takes at most n comparisons. Since we are merging k lists, we get k times n comparisons.

   b. Give pseudo code for a divide and conquer algorithm that solves the problem and show it is more efficient. For simplicity of the analysis of complexity you should assume that both n and k are powers of 2. Thus let $n = 2^s$ and $k = 2^t$. Note that this also implies that n/k is 2^{s-t}**.**

   Combine two lists at a time, with the same length, until only one list remains, using:

   Merge(A[], left, right, mid) left = first list, right = second list

```
        I = 0

        J = 0

        K = 0

        While (I < left.length and j < right.length)

                If left[i] <= right[j] then

                        A[k++] = left[i++];

                Else

                        A[k++] = left[j++];
```

```
While (I < left.length)
        A[k++] = left[i++];
While (J < right.length)
        A[k++] = right[j++];
Return A
```

Finding local min in nxn grid: Given an nxn grid, A of distinct numbers. A number is a local minimum if it is smaller than all its neighbors. That is A[i,j] is a local min if A[i,j] < min {A[i-1,j], A[i+1,j], A[i,j-1], A[i,j+1]} if 1 < i,j < n, For side and corner points there are only 3 and 2 numbers to compare to A[i,j] respectively. Use the divide and conquer paradigm to find a local minimum with only O(n) comparisons. Note there are n^2 numbers in the array so you cannot check all the numbers.

- Finding a solution for this problem can be quite difficult. In many problems it is a good idea to develop a simpler version of the problem and see if solving that version gives some insight in how to attack the more difficult version.
- A simpler version of this problem is the one dimensional **(1-D)** version: Given an array of distinct integers find a local minimum. At first glance this is trivial, just do a linear search for a global minimum and this is certainly guaranteed to be a local minimum.
- On the other hand, revisiting the original problem, it wants a O(n) solution with n^2 numbers. This means that the **1-D** version with n numbers should be faster than O(n). In addition, the problem is framed as being solvable using divide and conquer.

7. **Find a local min in a 1 dimensional array of integers:** Use divide and conquer to define an algorithm that solves this problem in O(log n) number of comparisons.

```
FindLocalMin(A[], low, high)
        If high > low
                Return false
        Mid = (low + high) / 2
        If (mid == 0 and A[mid] < A[mid + 1])
                Return true
        Else if(mid == 0)
                Return false
        If (mid == A.length -1 and A[mid] < A[mid - 1])
                Return true
        Else if(mid == A.length - 1)
                Return false
        If (A[mid] < A[mid + 1] and A[mid] < A[mid - 1])
                Return True
        If (A[mid] > A[mid + 1])
                Return FindLocalMin(A[], mid + 1, high)
        Return FindLocalMin(A[], low, mid)
```