4CCS1DST – Data Structures

Lecture 3 – Solutions to Exercises

Exercise 1: asymptotic notation

$10\,n$ is:	**O(n)**	**$O(n^2)$**	O(log n)	$\Theta(n^2)$	**$\Theta(n)$**
$n/2 + 5 \log n$ is:	**O(n)**	**$O(n^2)$**	$\Theta(n \log n)$	$\Theta(n^2)$	**$\Theta(n)$**
$7n^3 - 9n^2$ is:	O(n)	$O(n^2)$	**$\Omega(n^2)$**	O(log n)	$\Theta(n^2)$	**$\Theta(n^3)$**

Circle all correct answers.

Exercise 2

The following Java method determines whether the elements in a given range of array arr are all unique.

```
public static boolean isUniqueLoop(int[] arr, int start, int end) {
   for ( int i = start; i < end; i++ )
      for ( int j = i+1; j <= end; j++ )
         if ( arr[i] == arr[j] )
            return false;     // the same  element at locations i and j
   // all elements are unique
   return true;
}
```

What is the worst-case running time of this method, in terms of the number n of elements under consideration (n = end − start + 1) ?

Is there a better (faster) way to find out if all elements are unique?

Exercise 2 (cont.)

Answer:

The running time of this method is $O(n^2)$: at most n iterations of the outer loop, and at most n iterations of the inner loop for each iteration of the outer loop.

In the worst case (when all elements are unique), the total number of iterations of the inner loop is equal to

$$(n-1) + (n-2) + \ldots + 2 + 1 = \Theta(n^2),$$

so the worst case running time is $\Theta(n^2)$.

A faster algorithm for checking if the elements in the array are all unique:

Sort the array ($O(n \log n)$ time, by, for example, Merge Sort) and check the pairs of the consecutive elements ($O(n)$ time). The elements in the array are all unique if, and only if, each pair of two consecutive elements (after sorting) consists of two different numbers. The total running time of this algorithm is:

$$O(n \log n) + O(n) = O(n \log n).$$

Exercise 3

The following Java method determines whether three sets of integers, given in arrays a, b and c, have a common element.

```
public static boolean haveSameElement(int[] a, int[] b, int[] c) {
    for ( int i=0; i < a.length; i++ )
        for ( int j=0; j < b.length; j++ )
            for ( int k=0; k < c.length; k++ )
                if ( (a[i] == b[j]) && (b[j] == c[k]) )
                    return true;     // a common element found
    // no common element
    return false;
}
```

What is the worst-case running time of this method, if each array is of size n ?

Is there a better (faster) way to find out if the arrays have a common element?

Exercise 3 (cont.)

Answer:

The running time of this method is $O(n^3)$: three nested loops; each loop has at most n iterations.

In the worst case (when the three arrays do not have a common element), all three loops have n iterations. Hence the worst-case running time is $\Theta(n^3)$.

A faster algorithm:

Sort each array – this takes $O(n \log n)$ time.

Keep moving simultaneously along each array: in each step advance to the next position in the array where the current element is the smallest. If the three arrays have a common element, then the first (smallest) element will be discovered during this process. This process takes $O(n)$ time. Example:

a = {**3**, 5, 8, 15, 19}; b = {**2**, 6, 7, 9, 15}; c = {**1**, 5, 15, 16, 22}
a = {**3**, 5, 8, 15, 19}; b = {**2**, 6, 7, 9, 15}; c = {1, **5**, 15, 16, 22}
a = {**3**, 5, 8, 15, 19}; b = {2, **6**, 7, 9, 15}; c = {1, **5**, 15, 16, 22} ….

The total running time of this algorithm is $O(n \log n) + O(n) = O(n \log n)$.

Exercise 4

Design the following algorithm and implement it as a Java method

Algorithm *countOnes*(A, n)
Input two-dimensional n x n binary array A (each entry is either 0 or 1)
Output two-dimensional n x n array S, where $S[i][j]$ is the number of 1's in the "subarray" with the top-left corner at (0,0) and the bottom-right corner at (i, j).

Example. Input:

1	0	1	1	0
0	1	1	1	0
1	0	1	0	1
1	1	0	0	1
0	0	1	1	0

Output:

1	1	2	3	3
1	2	4	6	6
2	3	6	8	9
3	5	8	10	12
3	5	9	12	14

What is the running time of your algorithm in terms of n ?

Try to obtain the running time as low as you can.
The target running time is $\Theta(n^2)$.

Exercise 4 (cont.)

Answer:

```
public static int[][] countOnes(int[][] A, int n) {
    int[][] S = new int[n][n];
        for (int i = 0; i < n; i++) {
            for (int j = 0; j < n; j++) { S[i][j] = countOnesEntry(A,i,j); }
        }
        return S;
}

public static int countOnesEntry(int[][] A, int i, int j) {
        // count ones in the i-th row, columns 0, 1, ..., j
        int x = 0;
        for (int ii = 0; ii <= i; ii++) {
                for (int jj = 0; jj <= j; jj++) { x += A[ii][jj]; }
        }
        return x;
}
```

Exercise 4 (cont.)

The running time of the method countOnesEntry is $O(n^2)$.

The running time of the method countOnes is:

$$n^2 \text{ x } O(n^2) \;=\; O(n^4).$$

The running time of the methd countOnes is also $\Omega(n^4)$:

for each $n/2 < i < n$ and each $n/2 < j < n$, the computation of the entry S[i][j] takes $\Omega(n^2)$ time, and there are $n^2/4$ such entries.

Exercise 4 (cont.)

A faster method (other solutions possible):

```
public static int[][] countOnesFast(int[][] A, int n) {
    // assume that n >= 1
    int[][] S = new  int[n][n];
    S[0][0] = A[0][0];

    // first row
    for (int j = 1; j < n; j++) { S[0][j] = S[0][j-1] + A[0][j]; }

    // first column
    for (int i = 1; i < n; i++) { S[i][0] = S[i-1][0] + A[i][0]; }

    // the remaining entries in the array
    for (int i = 1; i < n; i++) {
       for (int j = 1; j < n; j++)
          { S[i][j] = S[i-1][j] + S[i][j-1] - S[i-1][j-1] + A[i][j]; }
    }
    return S;
}
```

The running time: $\Theta(n^2)$