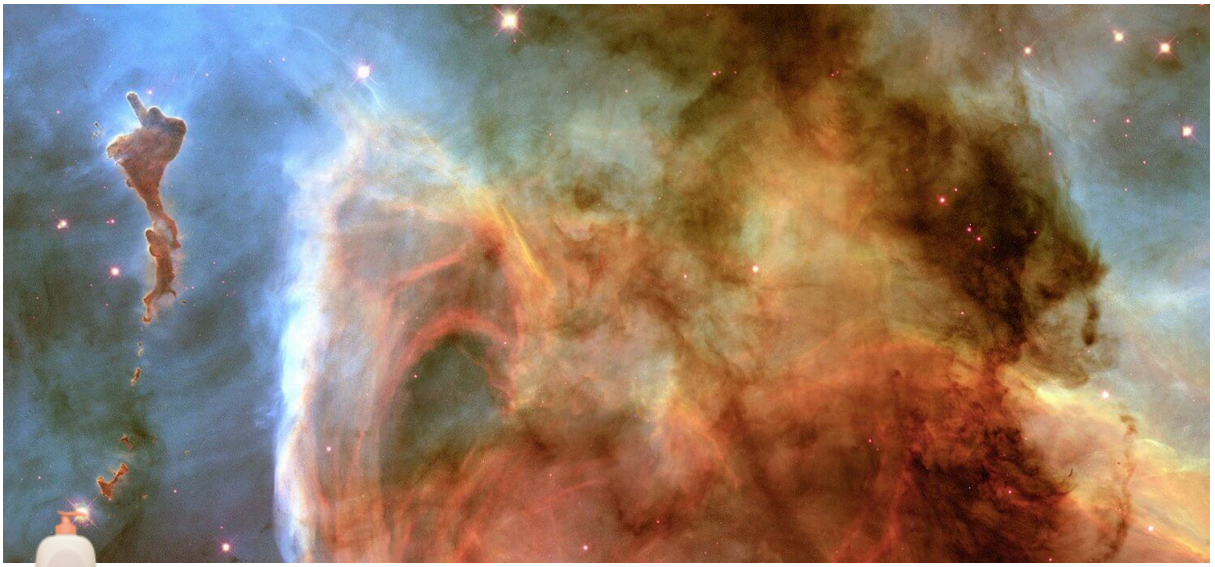

Solutions

Data Structures & Algorithms Labs
Week 2: Complexity and Efficiency

These exercises will allow you to test your knowledge of the lecture material from week 1 (complexity, efficiency, and search).

1 Programming Exercises

Once you have solved each of these exercises, try to figure out the (average/worst case) time and space complexity of your solution, and whether more efficient solutions exist.

► **1.1.** You are given a 2D integer matrix, that is an array of arrays of integers. It has size `m*n` where `m` is the number of arrays in the matrix and `n` is the size of each array (`matrix[i]`). Each array in the matrix is sorted in non-decreasing order. The arrays are also non-overlapping, in the sense that the last element of each array is not greater than the first element of the next (`matrix[i][m-1]` $\leq$ `matrix[i+1][0]` for all `i` in range). Create a function that that takes as input the given matrix and an integer target, and will return whether the matrix contains the target. ◄

Solution. If you think carefully about the constraints put on the matrix in the question, then hopefully you will realise that it is equivalent to a sorted 1D array. That is, if we were to copy the elements of this matrix row-by-row into an array of size `mn`, then this array would be sorted. The binary search solution below uses that fact to search the matrix as if it were a 1D array (with modular arithmetic), but without having to actually copy the elements into an array as this would be inefficient.

```
// Quadratic linear search solution: O(mn)
boolean searchMatrix(int[][] matrix, int target) {
    for (int i = 0; i < matrix.length; i++) {
        for (int j = 0; j < matrix[i].length; j++) {
            if (matrix[i][j] == target) {
                return true;
            }
        }
    }
    return false;
}

// An alternative to the below would be to search
// through all arrays to see if there is one
// which can potentially contain the target
// ie; matrix[i][0] <= target <= matrix[i][n-1]
// and then perform binary search on that array
// hence leading to an O(m + log (n)) solution.

// Binary search solution: O(log (m * n)) = O(log(m) + log(n))
boolean searchMatrix(int[][] matrix, int target) {

    int n = matrix[0].length;
    int m = matrix.length;

    int l = 0;
    int r = n*m - 1;

    while (l <= r) {
```

```
        int mid = l + (r-l)/2;
        int elem = matrix[mid/n][mid%n];

        if (elem < target) {
            l = mid + 1;
        } else if (elem > target) {
            r = mid - 1;
        } else {
            return true;
        }
    }

    return false;
}
```

Time complexity: $O(\log(mn))$, Space complexity: $O(1)$. □

► **1.2.** (Hard) repeat the previous exercise, but now do not assume that the arrays cannot overlap. Instead assume that both the rows and columns of the matrix are sorted, such that `matrix[i][j]` $\leq$ `matrix[i][j+1]` and `matrix[i][j]` $\leq$ `matrix[i+1][j]` for all `i` and `j` in range. ◄

Solution. This time the matrix is not equivalent to a sorted 1D array, so the previous solution will not work. The solution below loops over rows/columns (whichever is bigger) and then does a binary search over each row/column to see if it contains the target. This can be the most efficient if m is much smaller than n or vice versa. However, in the general case where both m and n are very large, there is a more optimal solution. Can you find it?

```
// Linear search over binary searches
boolean searchMatrix(int[][] matrix, int target) {

    int n = matrix[0].length;
    int m = matrix.length;

    if(n <= m) {
        for(int i = 0; i < n; i++) {
            int l = 0;
            int r = m-1;
            while (l <= r) {
                int mid = l + (r-l)/2;
                int elem = matrix[mid][i];

                if (elem < target) {
                    l = mid + 1;
                } else if (elem > target) {
                    r = mid - 1;
                } else {
                    return true;
                }
            }

        }
    } else {
        for(int i = 0; i < m; i++) {
            int l = 0;
            int r = n-1;
            while (l <= r) {
                int mid = l + (r-l)/2;
                int elem = matrix[i][mid];

                if (elem < target) {
                    l = mid + 1;
                } else if (elem > target) {
```

```
                    r = mid - 1;
                } else {
                    return true;
                }
            }
        }
    }

    return false;
}
```

Time complexity: $O(min(m,n)log(max(m,n))$, Space complexity: $O(1)$. □

► **1.3.** Given an integer x, $0 \leq x \leq 99999$, find its cube root without directly computing it (you can use Java's Math.pow function but only with integer exponent). Your answer must be within 10^{-3} of the exact answer. What if you instead allow x to be any floating point number in the given range? ◄

Solution. We can solve this by doing a binary search over floating point numbers up to a value x, each time comparing the cube of the float variable to x. This works a little differently to a binary search over integers since we don't increment or decrement the endpoints, that is we set `l = mid` not `mid + 1` and similarly for `r`. If we did increment/decrement then we would be skipping over some values. Note that we set the right endpoint initially to x, using the fact that $x^3 \leq x$ for any non-negative integer x. This would not work if x could be any floating point number, because $x^3 < x$ if $0 < x < 1$.

```
float searchCbrt(int x) {

    float l = 0;
    float r = x;

    while (l <= r) {
        float mid = l + (r-l)/2;
        float midCubed = mid*mid*mid;

        if(midCubed < x - 1e-3) {
            l = mid;
        } else if(midCubed > x + 1e-3) {
            r = mid;
        } else {
            return mid;
        }
    }
    return -1;
}
```

Time complexity: $O(log(x))$, Space complexity: $O(1)$. □

► **1.4.** (Hard) You are given an array of integers `a[0]`, ... `a[n-1]` and a positive integer `x`. You are allowed to perform the following operation as many times as you like: increase any element `a[i]` in the array by a positive integer amount. However, the sum of array elements cannot increase by more than `x`. Your goal is to maximise the smallest value in the array after performing the operations. Output the maximum possible value of the smallest element. ◄

Solution. This is very similar to the previous question: it is difficult to find the right answer, but easy to check if a guess is an over- or underestimate of the true answer. Let's say we think the answer is some value M. To check this we compute the change in the array sum if we took all the array elements that are less than M and replaced them with M. If the change in sum is less than `x` we have an underestimate, if it is greater than `x` we have an overestimate. We can use this to find the answer by binary search again, this time over integers between `aMin` and `aMin + x`, where `aMin` is the minimum entry in the input array.

```
int solution(int[] a, int x) {
    int n = a.length;
    if (n == 0) return 0;
    int aMin = a[0];
    for (int i = 1; i < n; i++) {
```

```
            aMin = Math.min(aMin, a[i]);
        }
        int l = aMin, r = aMin + x;
        while (l < r) {
            int mid = l + (r - l + 1) / 2;
            int sumChange = 0;
            for (int i = 0; i < n; i++) {
                sumChange += Math.max(mid - a[i], 0);
            }
            if (sumChange <= x) {
                l = mid;
            } else {
                r = mid - 1;
            }
        }
        return l;
}
```

Time complexity: $O(n \log(x))$, Space complexity: $O(1)$. □

2 Theory Exercises

In this section, the domain of all given functions is the set of positive integers: $\mathbb{Z}_+ = \{1, 2, 3, \ldots\}$. The following table evaluates some important functions of n for the first few values of n.

n	$\log_2 n$	$n \log_2 n$	n^2	n^3	2^n	n^n
1	0.00	0.00	1	1	2	1
2	1.00	2.00	4	8	4	4
3	1.58	4.75	9	27	8	27
4	2.00	8.00	16	64	16	256
5	2.32	11.61	25	125	32	3125
6	2.58	15.51	36	216	64	46656

A useful relation between different bases of logarithm is the following. Let $a, b, n > 0$. Then

$$\log_a n = \log_a b \cdot \log_b n.$$

To compare the growth of functions we often use the **Big-O notation**. This is defined as

$$f(n) \in O(g(n)) \iff \exists C, n_0 > 0 \text{ such that } \forall n \geq n_0 : f(n) \leq C \cdot g(n),$$

which (more verbosely) says that $f(n)$ is of order $g(n)$ if there exist two positive constants C and n_0 such that $f(n)$ is less than or equal to $Cg(n)$ for all n greater than or equal to n_0.

To show that $f(n) \in O(g(n))$ for a given pair of functions f, g, we need to find a pair of constants C, n_0, that satisfy the above definition. The pair C, n_0, need not be unique.

► **2.1.** In each of the following cases, find a constant $C > 0$ such that $f(n) < Cg(n)$ for all $n > 0$:

1. $f(n) = 2n^2 + 4$, and $g(n) = n^2$;
2. $f(n) = 3n^2 + 5n + 9$, and $g(n) = n^3$;
3. $f(n) = 3n \log_2 n + 5n + 8$, and $g(n) = n \log_2 n$. ◄

After you have done that show that $f(n) \in O(g(n))$.

Solution. 1. We do the following derivation.

$$\forall n \geq 1 : \quad f(n) = 2n^2 + 4 \leq 2n^2 + 4n^2 = 6n^2 = 6 \cdot g(n).$$

To obtain the inequality, we used the fact that $\forall n \geq 1$, we have $n^2 \geq n$, and thus $n^2 \geq 1$. Therefore, choosing $n_0 = 1$ and $C = 6$, satisfies the definition of Big-O notation, and $2n^2 + 4 \in O(n^2)$.

2. We do the following derivation.

$$\forall n \geq 1 : \quad f(n) = 3n^2 + 5n + 9 \leq 3n^3 + 5n^3 + 9n^3 = 17n^3 = 17 \cdot g(n).$$

To obtain the inequality, we used the fact that $\forall n \geq 1$, we have $n^2 \geq n$, and also $n^3 \geq n^2$, and thus $n^3 \geq n^2 \geq n \geq 1$. Therefore, choosing $n_0 = 1$, and $C = 17$, satisfies the definition of Big-O notation, and $3n^2 + 5n + 9 \in O(n^3)$.

3. We do the following derivation.

$$\forall n \geq 2 : \quad f(n) = 3n \log_2 n + 5n + 8 \leq 3n \log_2 n + 5n \log_2 n + 8n \log_2 n = 16n \log_2 n = 16 \cdot g(n).$$

To obtain the inequality, we used the fact that $\forall n \geq 2$, we have $\log_2 n \geq 1$, and also $n \log_2 n \geq n$, and thus $n \log_2 n \geq 1$. Note that this inequality does not hold for $n = 1$. Therefore, choosing $n_0 = 2$, and $C = 16$, satisfies the definition of Big-O notation, and $3n \log_2 n + 5n + 8 \in O(n \log_2 n)$. □

► **2.2.** Show that $\log_2 n \in O(\log_{10} n)$. Is it also true that $\log_{10} n \in O(\log_2 n)$? ◄

Solution. This is true, because $\forall n \geq 1$, we have $\log_2 n = \log_2 10 \cdot \log_{10} n \approx 3.3219 \log_{10} n$. Therefore, choosing $C = \log_2 10$ and $n_0 = 1$ satisfies the definition of Big-O notation, and $\log_2 n \in O(\log_{10} n)$. □

► **2.3.** Let f, g, h be functions taking on non-negative values. Show the following:

1. If there are constants $C, C' > 0$ such that $f(n) \leq Cg(n)$ for all $n > 0$, and $g(n) \leq C'h(n)$ for all $n > 0$, then there exists a constant $C'' > 0$ such that $f(n) \leq C''h(n)$.

2. If there are constants $C, C' > 0$ such that $f(n) \leq Ch(n)$ and $g(n) \leq C'h(n)$ for all $n > 0$, then there exists a constant $C'' > 0$ such that $f(n) + g(n) \leq C''h(n)$.

In both cases, express C'' in terms of C and C'. ◀

Solution. 1. Since $f(n) \in O(g(n))$, there exist $C_1, n_1 > 0$, such that

$$\forall n \geq n_1, \quad f(n) \leq C_1 \cdot g(n).$$

Also, since $g(n) \in O(h(n))$, there exist $C_2, n_2 > 0$, such that

$$\forall n \geq n_2, \quad g(n) \leq C_2 \cdot h(n).$$

Define $n_3 := \max\{n_1, n_2\}$. Combining the above two equations, we obtain

$$\forall n \geq n_3, \quad f(n) \leq C_1 \cdot g(n) \leq C_1 \cdot C_2 \cdot h(n).$$

Therefore, choosing $n_0 = n_3$, and $C = C_1 \cdot C_2$, satisfies the definition of Big-O notation, and $f(n) \in O(h(n))$.

2. Since $f(n) \in O(h(n))$, there exist $C_1, n_1 > 0$, such that

$$\forall n \geq n_1, \quad f(n) \leq C_1 \cdot h(n).$$

Also, since $g(n) \in O(h(n))$, there exist $C_2, n_2 > 0$, such that

$$\forall n \geq n_2, \quad f(n) \leq C_2 \cdot h(n).$$

Define $n_3 := \max\{n_1, n_2\}$. Combining the above two equations, we obtain

$$\forall n \geq n_3, \quad f(n) + g(n) \leq C_1 \cdot h(n) + C_2 \cdot h(n) = (C_1 + C_2) \cdot h(n).$$

Therefore, choosing $n_0 = n_3$, and $C = C_1 + C_2$, satisfies the definition of Big-O notation, and $f(n) + g(n) \in O(h(n))$. □

To show that $f(n) \notin O(g(n))$, one way is showing that no matter how we choose C and n_0, there is always some $n > n_0$, such that $f(n) > C \cdot g(n)$. More formally

$$f(n) \notin O(g(n)) \iff \forall C, n_0 > 0, \exists n > n_0 \text{ such that } f(n) > C \cdot g(n).$$

► **2.4.** In each of the following cases, show that $f(n) \notin O(g(n))$.

1. $f(n) = n^2$, and $g(n) = n$;
2. $g(n) = 5n^3$, and $g(n) = n^2$;
3. $f(n) = n \log_2 n$, and $g(n) = n$. ◀

Solution. 1. We first fix arbitrary $C, n_0 > 0$. We compare $f(n)$ and $C \cdot g(n)$, to find for which values of n, we have $f(n) > C \cdot g(n)$.

$$f(n) > C \cdot g(n) \iff n^2 > Cn \iff n^2 - Cn > 0 \iff n(n - C) > 0.$$

Now since $n > 0$, the factor $n - C$ must be positive, and thus, $n > C$. Note that this is valid, because $C > 0$. However, we also require that $n > n_0$. We can then choose any $n > \max\{C, n_0\}$. Therefore, for all $C, n_0 > 0$, we found an $n > n_0$ such that $f(n) > C \cdot g(n)$. Therefore, $f(n) \notin O(g(n))$.

2. We first fix arbitrary $C, n_0 > 0$. We compare $f(n)$ and $C \cdot g(n)$, to find for which values of n, we have $f(n) > C \cdot g(n)$.

$$f(n) > C \cdot g(n) \iff 5n^3 > Cn^2 \iff 5n^3 - Cn^2 > 0 \iff n^2(5n - C) > 0.$$

Now since $n^2 > 0$, the factor $5n - C$ must be positive, and thus, $n > C/5$. Note that this is valid, because $C/5 > 0$. However, we also require that $n > n_0$. We can then choose any $n > \max\{C/5, n_0\}$. Therefore, for all $C, n_0 > 0$, we found an $n > n_0$, such that $f(n) > C \cdot g(n)$. Therefore, $f(n) \notin O(g(n))$.

3. We first fix an arbitrary $C > 0$. We compare $f(n)$ and $C \cdot g(n)$, to find for which values of n, we have $f(n) > C \cdot g(n)$.

$$f(n) > C \cdot g(n) \iff n \log_2 n > Cn \iff n \log_2 n - Cn > 0 \iff n(\log_2 n - C) > 0.$$

Now since $n > 0$, the factor $\log_2 n - C$ must be positive, and thus, $n > 2^C$. Note that this is valid, because $2^C > 0$. However, we also require that $n > n_0$. We can then choose any $n > \max\{2^C, n_0\}$. Therefore, for all $C, n_0 > 0$, we found an $n > n_0$ such that $f(n) > C \cdot g(n)$. Therefore, $f(n) \notin O(g(n))$. □