Exercise 1. *1. Arrange the following functions in ascending order of growth rate (15 points):*

$$\left(\frac{3}{2}\right)^n, 100,\ n^3 \log^2 n,\ 2^{\log_2 n},\ \log^4 n,\ 2^{3\log_2 n},\ 2^n$$

Solution. The above functions would ordered as follows:

$$\begin{gathered} 100 \\ \log^4 n \\ 2^{\log_2 n} \\ 2^{3\log_2 n} \\ n^3 \log^2 n \\ \left(\frac{3}{2}\right)^n \\ 2^n \end{gathered}$$

Exercise 2. *Analyze the following code and provide a Big-O estimate of its running time in terms of n. Explain your analysis.*

Solution. Using the RAM model of analyzing runtime, we can address the algorithm as follows:

```
void compute (int n, double[][]A, double[][] B, double[][] C,
  ↪ double[][] D) {
    for (int i = 0; i < n; i++) { // n times
        for (int j = 0; j < n; j++) { // n times
            C[i][j] = A[i][j] + B[i][j]; // 2 operations
        }
    }

    for (int i = 0; i < n; i++) { // n times
        for (int j = 0; j < n; j++) { // n times
            for (int k = 0; k < n; i++) // n times
                D[i][j] = D[i][j] + A[i][k] * B[k][j]; // 3
                   ↪ operations
        }
    }
}
```

From the above comments, we can see that the runtime of the program will be $n \cdot n \cdot 2 + n \cdot n \cdot n \cdot 3 = 3n^3 + 2n^2$. Since the highest order term is n^3, we know that $3n^3 + 2n^2 \in \mathcal{O}(n^3)$.

Exercise 3. *Consider the following problem:*

Input: *an array, A, of n sorted integers such that*

$$A[0] \leq A[1] \leq A[2] \leq \ldots \leq A[n-2] \leq A[n-1]$$

Output: *re-arrange the elements in A such that:*

$$A[0] \leq A[1] \geq A[2] \leq A[3] \geq A[4] \leq \ldots$$

that is, even-indexed elements are less than their adjacent elements and odd-indexed elements are greater than their adjacent elements.

Design an algorithm that solves this problem in $\mathcal{O}(n)$ time. Analyze the algorithm using the RAM model.

Solution. We know that for any three elements of A given that $0 < i < n-1$, $A[i-1] \leq A[i] \leq A[i+1]$ implies $A[i-1] \leq A[i+1] \geq A[i]$. Thus, in order to achieve the output condition, we can perform the swap described for odd-indexed elements of A since the swap makes $A[i+1]$ greater than both of its neighbors while moving it to the ith position (which is odd-indexed). Pseudocode for the algorithm is as follows:

```
procedure swap(A[i], A[j]): // 3 operations
    temp = A[i] // 1 operation
    A[i] = A[j] // 1 operation
    A[j] = temp // 1 operation

procedure main:
    for i in odd integers from 1 to n-2: // approx. n/2 operations
        swap(A[i], A[i+1]) // 3 operations
```

We can see that the runtime of the algorithm, using RAM, would be $n/2 * 3 = 3n/2$. Since the highest order term is n, we can see that $3n/2 \in \mathcal{O}(n)$

Exercise 4.

Solution. As given in the problem, the element at index 0 will serve as the "pivot". Starting from the element at index 1, we want to put all values less than or equal to than the pivot on the left side and the elements greater to the right. First, we do this by checking the left side of the array, if an element is less than or equal to the pivot, we keep it on the left, if it is greater than the pivot, we swap it with an element on the right.

We keep track of elements on the right side so that once an element in swapped to the right, it does not get swapped again (since it is greater than the pivot). We repeat this process until the left side and the right side meet. We finish the algorithm by swapping the pivot if it is not already in the correct position.

```
// input: array of integers A

procedure swap(A[i], A[j]): // 3 ops
    temp = A[i] // 1 op
    A[i] = A[j] // 1 op
    A[j] = temp // 1 op

procedure main:
    pivot = A[0] // 1 op
    i = 1 // 1 op
    j = size(A) - 1 // 1 op
    while i < j:
        if A[i] <= pivot:
            i += 1 // 1 op
        else:
            swap(A[i], A[j]) // 3 ops
            j -= 1 // 1 op
    if size(A) > 1 and pivot > A[i]
        swap(A[0], A[i]) // 3 ops
    else
        swap(A[0], A[i-1]) // 3 ops
```

For estimation purposes, let us suppose that each side of the branch happens half of the time. We can also see that the i and j start on opposite sides of the array and will be incremented at most approximately n times before the while-loop breaks. This gives us a running time of $1 + 1 + 1 + n \cdot ((1)/2 + (4)/2) + 3$. We can see that n is the highest order term, and thus, the algorithm has a time complexity of $\mathcal{O}(n)$.