D0012E - Lab 2

December 4, 2025

Bardia Baigy: barbai-1@student.ltu.se
Vachana Javali: vacjav-2@student.ltu.se

Task 1

Given an array A of n => 4 distinct real numbers, determine if A contains three distinct elements x, y, z such that x+y=z. We assume n=2^k for k => 2. We use two recursive algorithms using an incremental approach, one which takes O(n^3) and another which takes O(n^2).

1.1 $O(n^3)$

Base case: If the input array A has fewer than 3 elements, return false (since you need three distinct elements x, y and z for computing the sum).

Recursive step (incremental): Firstly, let A_{n-1} be the subarray consisting of the first n - 1 elements of A, and let a_n be the last element. Then, recursively call the algorithm on A_{n-1}. If the recursive call returns true, return true. If the call returns false, check if the new element, $a_{n,}$ completes a x + y = z triplet with any two elements from A_{n-1}. Iterate through all pairs of a in A_{n-1} where i =!j. For each pair, check the three possible conditions for a triplet involving a_n : does $a_n + a_j = a_i$, $a_i + a_i = a_n$, $a_i + a_n = a_j$? If any of these conditions are met then return true.

Final result: If none of the recursive calls or pairwise checks returns true, the algorithm returns 0, meaning no such triplet exists.

1.2 $O(n^2)$

For this, we must ensure that each step is performed incrementally and that the time complexity for each increment is O(n). We do this by maintaining the first n elements in a sorted state which then allows the triplet check to be performed in O(n) time using the two-pointer technique.

Base case: If the input array A has less than 3 elements, return false.

Recursive step (incremental): Recursively call the algorithm on the subarray of size n-1, A[n-2]. This will ensure that A is sorted in place upon return and also returns any triplets found within the subarray. If returns true, return true.

Incremental sort(O(n)): Set element a_n = A[n-1] as the new element to be inserted. Do insertion sort, as in shift the elements in the sorted A[n-1] to the right until a_n is correctly positioned. Then using two pointers, check if the current last element A[n-1] can be formed by the sum x and y from the previous n-1 elements. Again using two pointers, check whether there exist two distinct elements x and y such that $x + a_n = y$ which is equivalent to checking $y-x = a_n$. If either case succeeds, return true.

Final result: If neither the recursive call nor the incremental checks find a valid triplet, return false.

Task 2

Maximum Subarray Sum using Divide and Conquer

Given an array "A" of non-zero real numbers, find the subarray(of consecutive elements) with the maximum possible sum. We will use a divide and conquer algorithm that runs in O(n) time. We assume that n is a power of 2.

Divide and Conquer Algorithm:

The maximum subarray sum can be in one of three places:

1. Within the **left half** of the array.
2. Within the **right half** of the array.
3. **Crossing the midpoint** (i.e., spans from the left half to the right half, including the elements adjacent to the midpoint).

The core of the O(n) solution is that finding the maximum crossing subarray sum can be done in O(n) time (where n is the size of the current subproblem).

Base Case: If the subarray has only one element (n=1), return that single element's value as total sum, starting index of subarray, ending index of subarray and best sum.

Divide: Find the midpoint m of the current subarray A[l...r].

Conquer (Recursive Calls):

1. Recursively find the maximum subarray sum in the left half A[l...m].
2. Recursively find the maximum subarray sum in the right half A[m+1...r].

Combine (Maximum Crossing Sum):
A. Compute prefix and suffix for subarray and check if array crosses over from left half to right half.

B. Compute the best sum by checking if it is in the left half, right half or in combination.

Return: Return total sum, starting index of subarray, ending index of subarray and best sum.

Task 3

Description of sortR():

The algorithm takes an array of n elements, A = <a_1, a_2, …, a_n>, and outputs a sorted array of the elements by making recursive calls to the function. The base case for the algorithm is the number of elements, n is ≤ 4. The elements are then sorted using insertion sort.

In the first recursive sort, the first three quarters of the input array, $<a_1, a_2, \ldots, a_{3n/4}>$, are recursively sorted giving $<b_1, b_2, \ldots, b_{3n/4}>$. In the second recursive sort, the last two thirds of the array from the first recursive sort, $<b_{n/4+1}, b_{n/4+2}, \ldots, b_{3n/4}>$, is combined with the last quarter from the input array $<a_{3n/4+1}, a_{3n/4+2}, \ldots, a_n>$ and sorted to give an array $<c_{n/4+1}, c_{n/4+2}, \ldots, c_n>$. In the third recursive sort, the first quarter of the sorted array from the first recursive sort, $<b_1, b_2, \ldots, b_{n/4}>$ and the first two thirds from the sorted array from the second recursive sort, $<c_{n/4+1}, c_{n/4+2}, \ldots, c_{3n/4}>$ are combined and sorted to give an array $<d_1, d_2, \ldots, d_{3n/4}>$. In the last recursive sort, the sorted array from the previous recursive sort $<d_1, d_2, \ldots, d_{3n/4}>$ is combined with the last quarter of the sorted array from the second recursive sort $<c_{3n/4+1}, c_{3n/4+2}, \ldots, c_n>$ to give a sorted array $<d_1, d_2, \ldots, d_{3n/4}, c_{3n/4+1}, c_{3n/4+2}, \ldots, c_n>$.

In conclusion, in every recursive sort call, the length of the array used for computation is 3n/4. The array used for computation is three quarters of the sorted array from the previous call and a quarter of the array yet to be combined and sorted. Basically, the input array is sorted by repeatedly sorting overlapping subarrays and the base case is solved by insertion sort.

Correctness of sortR:

We are going to prove that **sortR** gives a sorted array which is a permutation (P(n)) of a given input array.

Base case:

The algorithm uses insertion sort for $n \leq 4$ which is a standard sorting algorithm. The output is a sorted array consisting of the elements from the input array in ascending order. P(n) thus holds true for all $n \leq 4$.

Induction hypothesis:

We assume that **sortR** holds true for all array sizes $m < n$, i.e., it returns a sorted permutation of the input. We must now prove that **sortR** holds true for array size n.

Induction step:

Each recursive call computes on a subarray of size 3n/4 which is < n. Therefore, according to the induction hypothesis, each recursive call returns a correctly sorted array.
First recursive call: $<b_1, b_2, \ldots, b_{3n/4}> = 3n/4$
Second recursive call: $<c_{n/4+1}, c_{n/4+2}, \ldots, c_n> = 3n/4$
Third recursive call: $<d_1, d_2, \ldots, d_{3n/4}> = 3n/4$

For the final output we must show that the d array $<d_1, d_2, \ldots, d_{3n/4}>$ is $\leq$ the c array $<c_{n/4+1}, c_{n/4+2}, \ldots, c_n>$. We know from the previous step that array c is sorted correctly so then $<c_{n/4+1}, c_{n/4+2}, \ldots, c_{3n/4}>$ contains the smaller elements of c and $<c_{3n/4+1}, c_{3n/4+2}, \ldots, c_n>$ contains the larger elements of c. The d array consists of elements from the first quarter of b and the first half of c. This implies that $d \leq c$ since d consists of the smaller elements of c and c is made up of only the largest elements.

Thus, <d_1, d_2, …, $d_{3n/4}$, $c_{3n/4+1}$, $c_{3n/4+2}$, …, c_n> is a permutation of the input array and is correctly sorted for array size n.

Since the algorithm holds true for the base case and the inductive step, by mathematical induction the algorithm is true for all valid input sizes.

Recurrence equation for worst-case running time:

Let T(n) = worst case running time of sortR on an array of length n.

For the base case:

$$T(n) = O(1) \qquad \text{for } n \le 4$$

Since insertion sort is called on at most 4 elements so its cost is some fixed constant.

For the recursions:

Each recursive call computes an array of size 3n/4 which takes **T(3n/4)** time. Since there are three recursive calls this takes **3T(3n/4)** time in total. For the non-recursive work in the function (building arrays of size 3n/4 and concatenation of size n) the time taken is proportional i.e., **O(n).** In total, for the recursions the time taken is:

$$T(n) = 3T(3n/4) + O(n) \qquad \text{for } n > 4$$

Total recurrence:

$$T(n) = \begin{cases} O(1), & \text{for } n \le 4 \\ 3T(3n/4) + O(n) & \text{for } n > 4 \end{cases}$$

Solving the recurrence equation:

We can rewrite the recurrence equation as:

$$T(n) = \begin{cases} O(1), & \text{for } n \le 4 \\ 3T(3n/4) + cn & \text{for } n > 4 \end{cases}$$

Where c > 0. To solve the recursive case, we can rewrite the recurrence further in ***Master Theorem*** form:

$$T(n) = aT(n/b) + f(n) \qquad \text{for } n > 4$$

Where a ≥ 1, b > 1.
In this case, a = 3, b = 4/3 and f(n) = cn = O(n).

To compare growth rates, we need to calculate the critical exponent $n^{\log_b a} = n^{\log_{4/3} 3}$. The growth rate of f(n) = O(n) which can be rewritten in terms of the recursive part as $n^{\log_{4/3} 3 - \varepsilon}$ for some ε > 0. This tells us that the growth rate of the non-recursive part $O(n^{\log_{4/3} 3 - \varepsilon})$ < O(n) which according to the Master Theorem means that the recursive part dominates. Thus,

$$T(n) = O(n^{\log_{4/3} 3}) = O(n^{3.82}).$$