Problem 1: Majority Element

You are given as an input a list of n objects $A = [a_1, a_2, ..., a_n]$ where n is a power of 2. In $O(1)$ time you can check if two objects are the same or not. We say there is a majority element if there is an element which appears in the list A more than $n/2$ times. Give a **divide and conquer algorithm** to check if there is a majority element. Your algorithm should run in $O(n \log n)$ time.

Note, for two objects a_i and a_j you can check in constant time if $a_i == a_j$? But you cannot order them such as $a_i < a_j$ or $a_i > a_j$, that does not make sense with these objects (think of them as new symbols). **Hence, you cannot sort these objects or find their median.** Explain your algorithm in words, and analyze the running time of your algorithm including stating the recurrence. And explain why your algorithm is correct.

Note, there is an $O(n)$ time algorithm for this problem, but you will not receive extra credit for that solution, so we suggest aiming for the simpler $O(n \log n)$ time algorithm.

Solution.

First, note that if there is a majority element in the entire array, then it must be the majority element in either the first half of the array or the second half of the array (or both). This suggests the following divide and conquer algorithm. We first divide the array into 2 halves: the left half and the right half. We recursively find the majority element in the left half, if there is one, and in the right half, if there is one. Let ℓ denote the majority element in the left half and r the majority in the right half. We then scan through the entire array A and count the number of occurrences of ℓ and the number of occurrences of r. If one of these two candidates occurs $> n/2$ times then we return it as the majority element of the entire array, otherwise we return NULL.

Solution.

Running time: The running time of this algorithm satisfies the recurrence: $T(n) = 2T(\frac{n}{2}) + O(n)$, since there are two recursive calls on half the input, plus $O(n)$ work to determine which candidate element (if any) is the majority element. This recurrence solves to $O(n \log n)$, as desired.

Since a majority element must appear $> n/2$ times there can be ≤ 1 majority element. Note that any element that appears $\leq n/4$ times in the left half and $\leq n/4$ times in the right half will appear $\leq n/2$ times in the entire array and hence is not a majority element. Therefore, if there is a majority element m in the entire array A then it must be a majority element in the left half or in the right half, or in both halves. So the only possible majority element for the entire array A are the 2 majority elements of the left and right halves, this is why the algorithm is correct.

Problem 2: Neighboring-product sum
Given a list of n positive integers $A = [a_1, ..., a_n]$, we are looking for the largest neighboring-product sum. Adjacent elements can be multiplied together (but don't have to) and then we sum up. Each element can be matched with at most one of its neighbors.
Here are some examples.
Given the list $A = [1, 4, 2, 3, 7, 4, 1]$, the max neighboring-product sum is

$$41 = 1 + (4 \times 2) + 3 + (7 \times 4) + 1$$

Note, in this example you cannot use something like $(4 \times 2 \times 3)$ since 2 can get paired with at most one neighbor. Given $A = [2, 2, 1, 3, 2, 1, 2, 2, 1, 2]$ the max neighboring-product sum is

$$19 = (22) + 1 + (32) + 1 + (22) + 1 + 2$$

Hence, we pair some elements with one of their neighbors, these pairs are multiplied together, and then we add up the new numbers (either original unpaired elements or product of paired elements).
Design a dynamic programming algorithm to find the max neighboring-product sum.
(Faster (and correct) algorithm in $O(\cdot)$ notation is worth more credit.)

(a) Define the entries of your table in words. E.g., $T(i)$ is ..., or $T(i, j)$ is ....
Hint: Your table T should be one-dimensional.

Solution.

$T(i)$ is the maximum neighboring-product sum for elements $a_1, ...a_i$.

(b) State the recurrence for the entries of your table in terms of smaller subproblems. Hint: Express $T(i)$ in terms of $T(i1)$ and $T(i2)$.

Solution.

$T(i) = \max\{T(i-1) + a_i, T(i-2) + (a_i \times a_{i-1})\}$

(c) Write pseudocode for your algorithm to solve this problem.

Solution.

$T(0) = 0$
$T(1) = a_1$
`for` $i = 2 \to n$ `do:`
    $T(i) = \max\{T(i-1) + a_i, T(i-2) + (a_i \times a_{i-1})\}$
`return` $T(n)$

(d) Analyze the running time of your algorithm.

Solution.

There are $n + 1$ entries in table T, each of which requires constant work to compute (comparing two easily computable expressions). Therefore, the algorithm is $O(n)$.

Problem 3: Electoral College

The problem is to determine the set of states with the smallest total population that can provide the votes to win the electoral college. Formally, the problem is the following:
Let n be the number of states, p_i be the population of state i, v_i be the number of electoral votes for state i, and Z is the number of electoral votes needed to win. All electoral votes of a state go to a single candidate. The winning candidate is the one who receives at least Z electoral votes. You are given as input: $n; Z; p_1, ..., p_n$; and $v_1, ..., v_n$. Our goal is to find a set of states S with the smallest total population that has at least Z electoral votes in total. **You only have to output the total population of the set S, you do not need to output the set S itself.**
For example, if $n = 5$, populations $P = [200, 100, 30, 700, 250]$, votes $V = [5, 1, 2, 7, 6]$ and $Z = 12$ then the solution is 480 since $480 = 200+30+250$ and states $1, 3, 5$ have $5+2+6 = 13$ electoral votes.
Note in this example: $p_2 > p_3$ but $v_2 < v_3$, this might occur, but shouldn't affect your algorithm.
Design a dynamic programming algorithm for this problem.
(Faster (and correct) algorithm in $O(\cdot)$ notation is worth more credit.)

(a) Define the entries of your table in words. E.g., $T(i)$ is ..., or $T(i, j)$ is ....

Solution.

$T(i, j)$ is the minimum population needed from the first i states to win at least j electoral votes.

(b) State the recurrence for the entries of your table in terms of smaller subproblems.

Solution.

$$T(i,j) = \begin{cases} \min\{T(i-1,j), T(i-1,j-v_i) + p_i\} & \text{if } v_i \le j \\ \min\{p_i, T(i-1,j)\} & \text{if } v_i > j \end{cases}$$

(c) Write pseudocode for your algorithm to solve this problem.

Solution.

```
for i = 0 → n:  T(i,0) = 0
for j = 1 → Z:  T(0,j) = ∞
for i = 1 → n:
    for j = 1 → Z:
        if v_i ≤ j then T(i,j) = min{T(i-1,j), T(i-1,j-v_i) + p_i}
        else T(i,j) = min{p_i, T(i-1,j)}
return T(n,Z)
```

(d) Analyze the running time of your algorithm.

Solution.

There are nZ entries in table T, each of which requires constant work to compute (comparing two easily computable expressions). Therefore, the algorithm is $O(nZ)$.

Problem 4: FFT short answer
Let $\omega_n = e^{\frac{2\pi i}{n}}$ denote the n-th root of unity.
In polar coordinates $\omega_n = (1, \frac{2\pi}{n})$.

For all of the following assume $n = 2^k$ for integer $k \geq 2$, i.e., n is a power of two.

(a) What is $(\omega_n^{n/2})^{-1}$, in other words, what is the multiplicative inverse of $(\omega_n)^{n/2}$?
Let $(\omega_n^{n/2})^{-1} = \omega_n^j$. What is j? Specify a j where $0 \leq j \leq n$.

Solution.

B: $j = \frac{n}{2}$

$(\omega_n^{n/2}) = e^{\pi i} = -1$. The inverse of -1 is also -1, so the inverse of $(\omega_n^{n/2})$ is $(\omega_n^{n/2})$, and $j = \frac{n}{2}$.

(b) $\omega_n^0 + \omega_n + \omega_n^2 + \omega_n^3 + ... + \omega_n^{n-1} =?$

Solution.

E: 0

This is the sum of the n n-th roots of unity. As proved in the lecture the sum of the n-th roots of unity is 0. Another explanation: Each $(\omega_n)^j$ is located on the unit circle, and they are evenly spaced around it. Taking the average of their locations will be the center of the circle.

(c) $\omega_n^0 \times \omega_n \times \omega_n^2 \times \omega_n^3 \times ... \times \omega_n^{n-1} =?$

Solution.

E: -1

Note the inverse of ω_n^j is ω_n^{n-j} and hence $\omega_n^j \times \omega_n^{n-j} = 1$. Therefore in the above product for each term, except $j = 0$ and $j = n/2$, their inverse also appears and hence we have the following:

$$\begin{aligned}
&\omega_n^0 \times \omega_n \times \omega_n^2 \times \omega_n^3 \times ... \times \omega_n^{n-1} \\
&= \omega_n^0 \times \omega_n^{n/2} \times (\omega_n^1 \times \omega_n^{n-1}) \times (\omega_n^2 \times \omega_n^{n-2}) \times \ldots (\omega_n^{n/2-1} \times \omega_n^{n/2+1}) \\
&= \omega_n^0 \times \omega_n^{n/2} \times (1) \times (1) \times \ldots (1) \\
&= \omega_n^0 \times \omega_n^{n/2} \\
&= 1 \times -1 \\
&= -1.
\end{aligned}$$

Another way of seeing it: $\omega_n^0 \times \omega_n \times \omega_n^2 \times \omega_n^3 \times ... \times \omega_n^{n-1} = \omega_n^{\sum_{i=0}^{n-1} i} = e^{\frac{2\pi i}{n} \sum_{i=0}^{n-1} i} = e^{\frac{2\pi i}{n} \frac{n(n-1)}{2}} = e^{(n-1)\pi i}$. Since $e^{2\pi i} = 1$ and n is assumed to be even (more specifically, a power of 2), $e^{(n-1)\pi i} = e^{\pi i} = -1$.

(d) It is always true that the square-root of an n^{th} root of unity is a:

Solution.

A: $2n^{\text{th}}$ root

Note: $\omega_{2n} = (1, 2\pi/(2n)) = (1, \pi/n)$. Squaring it we have: $(\omega_{2n})^2 = (1, \pi/n) \times (1, \pi/n) = (1, 2\pi/n) = \omega_n$. Hence, the square of the $2n$-th roots are the n-th roots, or in other words, the square root of the n-th roots are the $2n$-th roots.

Another way of seeing it: $\sqrt{\omega_n} = \sqrt{e^{\frac{2\pi i}{n}}} = e^{\frac{2\pi i}{2n}} = \omega_{2n}$

(e) What is ω_n^k in polar coordinates (in terms of n and k)?

Solution.

E: $(1, \frac{2\pi k}{n})$

All roots of unity are on the unit circle, so they are at distance 1 from the origin and the first polar coordinate is 1. Each ω_n^k moves around the circle equally, with the final ω_n^n completing the circle. There are 2π radians in a circle, so ω_n is at $(1, \frac{2\pi}{n})$. Each successive ω_n^k is another $\frac{2\pi}{n}$ around the circle, so ω_n^k is at polar coordinates $(1, \frac{2\pi k}{n})$.

Problem 5: Recurrences

(a) The recurrence $T(n) = 4T(n/2) + O(n)$ solves to:

Solution.

D: $O(n^2)$

Since the recursive work dominates the $O(n)$ work, this recurrence solves to $O(n^{\log_2 4}) = O(n^2)$.

(b) The recurrence $T(n) = 2T(n/2) + O(1)$ solves to:

Solution.

B: $O(n)$

Since the recursive work dominates the $O(1)$ work, this recurrence solves to $O(n^{\log_2 2}) = O(n)$.

(c) The recurrence $T(n) = 3T(n/5) + O(n)$ solves to:

Solution.

B: $O(n)$

Since the $O(n)$ work dominates the recursive work, this recurrence solves to just $O(n)$.