Most of these problems are taken from exams during the years 2002-2012. They are partitioned into two categories. The first is composed of problems that are ready to be used. The second is composed of problems that have some problematic issues.
1 Ready Problems

1. Let \( A = [A[1], A[2], \ldots, A[n]] \) be an array containing all the numbers 1, \ldots, \( n \) (\( A \) represents a permutation). A sequence of indices \( \langle i_1, i_2, \ldots, i_k \rangle \) is a cycle of size \( k \) if

\[
\]

Example: If \( A[i] = i \) then the singleton sequence \( \langle i \rangle \) is a cycle of size 1.

Example: In the array \([4, 6, 3, 5, 8, 7, 2, 1]\) the sequence \( \langle 1, 4, 5, 8 \rangle \) is a cycle of size 4. The other 2 cycles are \( \langle 2, 6, 7 \rangle \) and \( \langle 3 \rangle \).

Design an efficient algorithm that finds the number of cycles in \( A \). What is the running time of your algorithm? Justify your answer.
2. Given two positive integers \( n \) and \( m \) the goal is to compute \( n^m \) with as minimum as possible integer multiplications.

(a) Describe a correct algorithm to compute \( n^m \) and analyze its complexity (number of multiplications).

Justify the correctness of your algorithm and explain your complexity analysis.

(b) Assume that \( m = 2^k \) is a power of 2 where \( k \geq 1 \) is a known integer.

Describe a correct and efficient algorithm to compute \( n^m \) and analyze its complexity (number of multiplications).

Justify the correctness of your algorithm and explain your complexity analysis.

(c) Describe a more efficient correct algorithm to compute \( n^m \) for any value of \( m \) and analyze its complexity (number of multiplications).

Justify the correctness of your algorithm and explain your complexity analysis.
3. Let \( A = [A_1 < A_2 < \cdots < A_n] \) and \( B = [B_1 < B_2 < \cdots < B_n] \) be two sorted arrays of integers of size \( n \geq 1 \). The integers in each array are distinct. However, there could be some integers that appear in both arrays. The goal in part (a) is to find one of the integers that appear in both arrays if such integers exist or to announce that the arrays are disjoint. The goal in part (b) is to find one of the integers that appear only in one of the arrays if such integers exist or to announce that the arrays are identical.

An algorithm may apply comparisons between an integer from \( A \) and an integer from \( B \). The output of a comparison is either the order among the two integers or stating that they are the same.

- In both parts, an efficient algorithm should make as few as possible comparisons between integers in the worst-case.
- For each part, justify the correctness of your algorithm and state the number of comparisons in the worst-case. A full credit will be given to an answer that states the exact number of comparisons. Partial credit will be given for answers using the big \( O \) notation.
- Partial credit will be given for inefficient but correct algorithms while no credit will be given for incorrect algorithms. If you are not sure about your efficient algorithm, you may provide a correct less efficient algorithm to get partial credit.
- Words are “better” than codes to describe algorithms.

(a) Describe an efficient algorithm to find an integer that appears in both arrays if exists. If such a number does not exist, the algorithm should output that both arrays are disjoint.

(b) Describe an efficient algorithm to find an integer that appears in only one of the arrays if exists. If such a number does not exist, the algorithm should output that both arrays are identical.
4. Let $S$ be a set of $n > 0$ distinct integers. Assume that $n$ is a power of 2. Here are some known results for the number of comparisons needed to execute some tasks on $S$:

- Finding the largest integer in $S$ is possible with $n - 1$ comparisons.
- Finding both the largest and smallest integers in $S$ is possible with $3n/2 - 2$ comparisons.
- Finding both the largest and second largest integers in $S$ is possible with $n + \log_2(n) - 2$ comparisons.

**Remarks:** In all the problems below you must explain why your algorithms are correct and state the exact number of comparisons they use in the worst case. Partial credit will be given for inefficient but correct algorithms while no credit will be given for incorrect algorithms. If you are not sure about your efficient algorithm, you may provide a correct less efficient algorithm to get partial credit. Words are “better” than codes to describe algorithms.

Assume now that $n > 0$ is a power of 3 and that each comparison can compare three integers from $S$ and order them from the largest to the smallest. Call such a comparison a ternary comparison.

(a) Describe an efficient algorithm that uses as few as possible ternary comparisons to find the largest integer in $S$.

(b) Describe an efficient algorithm that uses as few as possible ternary comparisons to find both the largest and the smallest integers in $S$.

(c) Describe an efficient algorithm that uses as few as possible ternary comparisons to find both the largest and the second largest integers in $S$. 

2 Problems with issues

1. (a) Let $S = S[1] < S[2] < \cdots < S[n]$ and $T = T[1] < T[2] < \cdots < T[n]$ be two sorted arrays of size $n$ containing $2n$ distinct numbers. Describe an $O(\log(n))$ algorithm to find the $n$th largest number among the $2n$ numbers. Explain why the algorithm is correct and why its complexity is $O(\log(n))$.

(b) Let $A$ and $B$ be two increasingly sorted arrays each of size $n$. Assume that all the $2n$ entries are integers and all of them are distinct. The goal is to partition the $2n$ numbers into two sets: one containing the largest $n$ integers and one containing the smallest $n$ integers. Describe an efficient algorithm that finds these sets with as minimum as possible comparisons between integers from both arrays. What is the worst case number of comparisons made by your algorithm as a function of $n$? Justify your answer. You may assume that $n$ is a power of 2 and may ignore floors and ceilings.
2. For $n \geq 1$ and $k \geq 1$, let $A$, $B$, and $C$ be three sorted arrays of integers:

\begin{align*}
C &= C[1] \leq C[2] \leq \cdots \leq C[k]
\end{align*}

Describe an efficient algorithm to decide if there exist two numbers one in $A$ and one in $B$ whose sum appears in $C$. Formally, decide if there exist three indices $1 \leq i, j \leq n$ and $1 \leq h \leq k$ such that $A[i] + B[j] = C[h]$.

What is the complexity of your algorithm as a function of $n$ and $k$? Justify your answer.

Remark: $k$ can be much larger or much smaller than $n$. As a result, for different ranges of $k$ and $n$ there could be different most efficient algorithms.
3. For \( n \geq 1 \), let \( A \) and \( B \) two arrays each containing the same set of \( n \) distinct integers. However the order among the integers in each array may be different.

Comparisons between integers one from each array are allowed however comparisons between two integers from the same array are forbidden. Each comparison either finds the larger integer and the smaller integer or outputs that both integers are the same.

(a) Describe an algorithm that finds the indices in \( A \) and \( B \) that contain the largest integer among the \( n \) integers with at most \( 2n - 2 \) comparisons.

Explain why your algorithm is correct.

(b) Prove that it is impossible to guarantee finding these indices with less than \( 2n - 2 \) comparisons.

(c) Describe an \( O(n^2) \) algorithm that sorts both arrays.

Prove that your algorithm is correct and that its complexity is \( O(n^2) \). Then explain why the worst-case complexity of your algorithm is \( \Omega(n^2) \).

(d) Describe a more efficient \((O(n \log(n)))\) (expected time) algorithm that sorts both arrays.

Explain why your algorithm is correct and justify the complexity claim.

**Hint:** It seems like it is easier to adapt Quick-Sort.
4. Let the numbers 1, 2, \ldots, n^2 be assigned to an \( n \times n \) matrix \( A \) such that each row is an increasing sequence of \( n \) numbers from left to right and each column is an increasing sequence of \( n \) numbers from top to bottom. Call such a matrix a “rows-columns-sorted-matrix.” Formally, for any row \( i \), if \( 1 \leq j < j' \leq n \) then \( A[i, j] < A[i, j'] \) and for any column \( j \), if \( 1 \leq i < i' \leq n \) then \( A[i, j] < A[i', j] \).

A prob to the matrix is a query “is \( A[i, j] = x \)?” The answer to a prob is “YES” or “NO”. The only way to access the matrix is via probs. Note that if the answer is “NO”, then you do not know if \( x < A[i, j] \) or \( x > A[i, j] \).

For a given \( x \), the goal is to find the two indices \( 1 \leq i, j \leq n \) such that \( A[i, j] = x \) with as few as possible probs.

(a) For any \( n \geq 2 \), show how to construct a rows-columns-sorted-matrix of size \( n \times n \).

(b) For an odd \( n \), construct a rows-columns-sorted-matrix \( A \) for which the median of the medians is not in the center of the matrix. In other words, the median \( m = \left\lceil \frac{n^2}{2} \right\rceil \) is not at location \( A[\lceil n/2 \rceil, \lceil n/2 \rceil] \). Try to find such an example for as small as possible odd \( n \).

(c) Describe efficient algorithms to find the locations of the numbers 1, 2, 3, 4 with as few as possible probs. What is the exact worst case number of probs used by your algorithm?

(d) Describe an efficient algorithm to find the location of a number \( k \leq n \) with as few as possible probs. What is the worst case number of probs used by your algorithm as a function of \( k \) and/or \( n \)? You may use the big-\( O \) notation. Explain why the algorithm is correct and justify your complexity statement.