MID-TERM

Remarks: Solve 4 of the following 5 questions. In all algorithm, always explain how and why they work. ALWAYS, analyze the complexity of your algorithms. In all algorithms, always try to get the fastest possible. A correct algorithm with slow running time may not get full credit.

Assumption: For questions 1, 2 you can use the following: There is an algorithm \( A \) finding the \( i \)-th smallest element in an \( n \) element array in \( O(n) \) time.

1. Question 1: You are given an array \( A[1..n] \) and a number \( k \). Give an algorithm that finds the \( k \) smallest, \( 2k \) smallest, \( 3k \) smallest, etc., elements in \( A[1..n] \).

Example: If \( A = <4, 8, 3, 9, 6, 7, 20, 1, 43> \) and \( k = 2 \) then the algorithm returns: 3 (the second smallest) 6 (the forth smallest) 8 (the six smallest) and 20 (the eight smallest).

Answer: Solution 1: Use the black box procedure, directly. For \( i = k \), \( i = 2k \), \( i = 3k \), etc. find the \( i \) smallest number (this is done for every \( i \), by running \( A \)). The number of times we run \( A \) is \( n/k \). Each time, it requires \( O(n) \) basic operations. This gives a total of \( O(n^2/k) \) running time.

Solution 2: Sort the array with merge-sort. This requires \( O(n \log n) \). Then, let \( A' \) be the sorted array. Return, \( \{A'[k], A'[2\cdot k], \ldots\} \). This requires \( O(n \cdot \log n) + O(n) = O(n \cdot \log n) \).

Solution 3: The best solution is: Run a partial version of Quicksort. In this version we use the median as pivot. The way to find the median is by using \( A \). Then, we partition around the median.

The partition takes \( O(n) \) and the median finding is \( O(n) \) so \( O(n) \) in total.

Now, we have 2 arrays \( A_s \) and \( A_t \) of size (roughly) \( n/2 \). Recursively, find the median in each one of them, and partition \( A_s \) and \( A_t \) using the partition procedure into four arrays \( A_{ss}, A_{sl}, A_{ts}, A_{tt} \) each of size about \( n/4 \).

Continue like this. The stopping condition in Quick sort is that we stop if the array has size 1. However, here we stop if the array has size \( k \).

At the end we will get \( n/k \) arrays each of size \( k \). Let \( A_1 \) be the \( k \) first elements of \( A \), \( A_2 \) the next \( k \) elements, etc. It is not hard to see that \( A_1 \) will have the \( k \) smallest elements of \( A \). \( A_2 \) will have the \( k+1 \) to \( 2k \) smallest elements in \( A \), \( A_2 \) the \( 2k+1 \ldots, 3k \) smallest elements in \( A \), etc. However, every one of the \( A_i \) is not sorted by itself. Thus, the \( k \) smallest element is the maximum in \( A_1 \). The \( 2k \) element is the maximum in \( A_2 \), etc.
This last step requires $O(k) \times O(n/k) = O(n)$ time (n/k arrays of size k). Each requiring $O(k)$ to find the maximum).

Running time: each level of the recursion tree in quicksort performs $O(n)$ basic operations. However, the number of levels is not $\log n$. We cut the arrays by 1/2 each level. The number of levels is the smallest $i$ so that $n/2^i \leq k$. Solving this gives $i = O(\log(n/k))$.

So, the running time is $O(n \log(n/k))$.

2. **Question 2:** Given an array $A[1..n]$ and a number $k$ give an algorithm that finds if there is a values that repeats at least $k$ time in the array.

**Example:** In $A =< 2, 4, 2, 5, 7, 2, 8, 2, 9 >$ and $k = 4$ the answer is yes. For $k = 5$ the answer is no.

**Answer:** Answer 1: Check for $i = 1$ to $n$ how many elements are equal to $A[i]$. Keep a variable that stores the maximum number of repetitions found. The running time is $O(n^2)$.


Then, we let $x \leftarrow A'[i]$ for the last $i$ we had in the previous loop (the first value different from $A'[1]$) and check again for the number of contiguous values equal to $A'[i]$, etc. This requires only $O(n)$ in total, as we are going over the array only once. The answer now is yes if a number repeating at least $k$ times is found. However, the total running time is $O(n \cdot \log n)$ (because of the mergesort run).

Best solution: Use the solution to question 1. Find the $k$ smallest elements in $A$, stored in $A_1$, the $k$ next smallest elements in $A$ stored in $A_2$, the next $k$ smallest elements in $A$, stored in $A_3$, etc.

A value that repeats at least $k$ times has to be either the maximum of $A_1$, or the maximum of $A_2$, or the maximum of $A_3$, etc (think about it if this is not clear).

Let $c \leftarrow \max\{a \mid a \in A_1\}$. If $c$ repeats $k$ times, this means that the value $c$ appears $k$ times or more in $A_1 \cup A_2$ (why?). So, check how many times $c$ appears in $A_1 \cup A_2$. Since $|A_1| + |A_2| = 2 \cdot k$, checking how many times $c$ appears there requires $O(k)$.

This is done in the same way for the max of $A_2$. We check how many times it repeats in $A_2 \cup A_3$, etc.

This, we have $n/k$ arrays, each $O(k)$ basic operations, to give $O(n/k) \cdot O(k) = O(n)$.

In total, like question 1, the running time is $O(n \log(n/k))$.

3. **Question 3:** We are given two arrays $A[1..n]$, $B[1..n]$. No value appears in $A$ more than once. No value appears in $B$ appears more than once. The arrays are not in any particular order. Give an algorithm that returns the set of value that appears both in $A$ and in $B$
**Answer:** Many students answer this question with solutions of algorithms whose running times are $O(n^2)$ answers. But, as I told you, sorting requires $O(n \cdot \log n)$ so compared to $O(n^2)$ its negligible. So, if you sort first, you do not lose. If you shoot for $O(n^2)$, try to see if $O(n \log n)$ is not possible by first sorting the array. In contrast, if you are “shooting” for an $O(n)$ algorithm, its not possible to sort.

Thus, sort $A$ and $B$. Let $A', B'$ be the sorted array. Like in “Merge” (of Mergesort) keep two indices on the last $A[i]$ of $A'$ that has been scanned, and an index on the last element $A[j]$ of $B$ that has been scanned.

At start $i = j = 1$. Now, if $A[i] = A[j]$, then add $A[i]$ to the set of values in the intersection, (if its not there already) and $i \leftarrow i + 1$, $j \leftarrow j + 1$.


Besides the sorting, we go over the arrays $A'$ and $B'$ once. Hence the running time is $O(n \cdot \log n) + O(n) = O(n \log n)$ (the $O(n \cdot \log n)$ comes from Mergesort).

4. **Question 4:** Consider the Knapsack problem, but under the assumption that elements have only two possible volumes (the prices can be arbitrary). Give an algorithm for this variant.

**Answer:** Since there are only 2 values, say, $v_1, v_2$ if we know the amount of items of volume $v_1$ in the solution, and the amount of items of volume $v_2$ in the solution, we know the best price of a legal solution. Because, if for example there are 4 items with volume $v_1$ in the solution, automatically they must be the four items with largest four prices among the items of $v_1$ volume.

Sort the items of volume $v_1$ by decreasing price. Let $P(1, i)$ be the sum of prices of the first $i$ items with largest price among those items with volume $v_1$. Sort the items of volume $v_2$ by decreasing price. Let $P(2, i)$ be the sum of prices of the first $i$ largest prices among items of volume $v_2$.

We say that $j(i)$ is the *largest legal index* for $i$ if $v_1 \cdot i + v_2 \cdot j \leq V$ ($V$ is the volume bound) but $v_1 \cdot i + v_2 \cdot (j + 1) > V$. This means that if $i$ items of volume $v_1$ are chosen, exactly $j$ items of volume $v_2$ are chosen. It is easy to see that:

$$j(i) = \left\lfloor \frac{V - iv_1}{v_2} \right\rfloor .$$

The algorithm:

(a) Let $Max \leftarrow -\infty$

(b) For $i = 0$ to $i = n$ do

i. Let

$$j(i) \leftarrow \left\lfloor \frac{V - iv_1}{v_2} \right\rfloor$$

be the largest legal index for $i$. 


ii. If $Max < P(1, i) + P(2, j(i))$ and $i \cdot v_1 \leq V$ then $Max \leftarrow P(1, i) + P(2, j(i))$
(c) Return $Max$

The algorithm goes over all possible numbers $i$ for the amount of items of volume $v_1$ in the best solution. If the number of items of volume $v_1$ in the optimum is $i$, then it must be that $j(i)$ items of volume $v_2$ are chosen. This is because $j(i)$ is the maximum possible.

If $i, j(i)$ are the correct pair, by definition their total price is $P(1, i) + P(2, j(i))$. Like in finding a maximum, the best solution is stored in $Max$.

It should be clear from the code that the running time of the procedure, except for the sorting is $O(n)$. However, the sorting renders it $O(n \log n)$ running time.

5. **Question 5:** In the pairs subset-sum problem you are given a collection of $n$ pairs of numbers $\{v_i, u_i\}_{i=1}^n$ and a number $S$. A feasible selection takes for every $i$ either $u_i$, or $v_i$ (but not both) or neither of $u_i$ and $v_i$. Give an algorithm to check if there is a feasible selection whose values sum to $S$. 