Solutions for Problem Set 1

Problem 1. Recurrence relations

Solution

All the exercises can be solved using the master theorem

(a)
In order to apply case 3 of the master theorem we have to prove the regularity of \( f(n) = n^2 \).
\( af(n/b) = 5 \frac{n^2}{16} \) that, \( \forall c \) such that \( \frac{5}{16} < c < 1 \), have the property that
\( af(n/b) \leq cf(n) \).
Thus the solution to the recurrence is \( T(n) = \Theta(n^2) \)

(b)
Master theorem case 1: \( T(n) = \Theta(n \log 3) \)

(c)
Master theorem case 2: \( T(n) = \Theta(n^3 \log n) \)

(d)
Master theorem case 1: \( T(n) = \Theta(n) \)

(e)
Like in exercise (a) we have to prove the regularity of \( f(n) = n^2 \).
This time \( af(n/b) = 3 \frac{n^2}{9} \) that \( \forall c \) such that \( \frac{3}{9} < c < 1 \) is such that
\( af(n/b) \leq cf(n) \).
Thus the solution to the recurrence is again \( T(n) = \Theta(n^2) \)

Problem 2. Divide and Conquer

(a)

Solution

Algorithm description

The following algorithm finds \( x \), if it exists, in the infinite array \( A \).

\( \infty \) or the value \( x \) is found. Let \( q \) be the last index that has been considered.
2. If \( x = A[q] \) then return \( q \).

3. Otherwise, do a binary search in \( A[q/2]...A[q] \) for \( x \). If \( x \) is found, return the index, otherwise, return FALSE.

**Correctness**

The first part of the algorithm narrows down the search to the portion of the array containing actual data. The binary search will find \( x \) (if it exists in the array) because the array is in sorted order (with \( \infty \) necessarily being greater than any integer in the array).

**Analysis**

The first part of the algorithm will take at most \( \log_2 q \) iterations, each one taking constant time, since at each iteration the value of the index is doubled, so that at the \( k \)-th iteration the value of the index is \( 2^k \) (and the value of the index is \( q \) at the \( \log_2 q \)-th iteration). We also have that \( n > q/2 \), otherwise the algorithm would have stopped when the index had the value \( q/2 \), so the number of iterations is at most \( \log_2 2n \) and the time taken by the first part of the algorithm is \( O(\log n) \). Binary search also takes \( O(\log n) \) time. So the algorithm as a whole takes \( O(\log n) \) time.

(b)

**Solution**

**Algorithm description**

We assume the existence of the following procedures:

- Merge-sort: sorts a vector of \( n \) integers in \( O(n \log n) \) time.

- Bin-search\((S, z, i, j)\) takes in input an integer vector \( S[] \), integer \( z \), and vector indices \( i < j \), and looks for the value \( z \) in the sequence \( S[i], S[i+1],..., S[j] \). The procedure returns TRUE if and only if \( z \) belongs to the sequence. The procedure takes \( O(\log(j-i)) \) time.

The proposed solutions works as follows:

1. Sort the array \( S \) using Merge-Sort.

2. Apply the following algorithm to the sorted array.

   **Search for Sums** \((S, n, x)\)
   
   \[
   \text{for } i = 1 \text{ to } n \text{ do}
   \]
   
   \[
   y = x - S[i]
   \]
   
   \[
   \text{if Bin-search}(S, y, i + 1, n) \text{ return YES.}
   \]
   
   \[
   \text{end for}
   \]
   
   return NO.

Where \( \text{Found}(S, z, j, k) \) is a binary search procedure that looks for \( z \) in \( S[i]...S[n] \):
Correctness
Let’s prove the correctness of Search for Sums (we already know that Merge-Sort and Bin-search are correct).

Correctness of Search for Sums
If there are two elements $z_1$ and $z_2$ in $S$ (w.l.o.g. suppose $z_1 > z_2$) such that $x = z_1 + z_2$ Search for Sums will answer YES.

By the hypothesis, will exist a $j \in [1, \ldots, n]$ such that $S[j] = z_1$. Then, by the correctness of Found, at the $j^{th}$ iteration the algorithm, will answer YES.

With a similar argument can be proved that the converse also holds.

Analysis
The time complexity of the proposed method is clearly given by:

Time complexity of Merge Sort ($O(n \log n)$) + Time complexity of Search for Sums ($n$-Time complexity Bin-search). Note that the time complexity of the procedure Bin-search is $O(\log n)$. Then the running time of the whole computation is bounded by $O(n \log n)$

Problem 3. Selection

(a)
Algorithm description
The following algorithm decides whether there is a value that is repeated $n/2$ times or more in the array:

1. Use the SELECT algorithm to find the median of the array.
2. Loop through the array and count the number of times that this value occurs.
3. If this value occurs more than $n/2$ times, return TRUE; Otherwise, return FALSE.

Correctness
Consider a sorted version of the array. Any item occurring more than $n/2$ times must cross the center of this sorted array, and therefore it would be the median of the array. So if the median of the array is not repeated $n/2$ times, than no element can be repeated that many times!

Analysis
As discussed in class and in CLR, the SELECT algorithm takes $O(n)$ time. Clearly, looping through the array to count the number of times that a particular value occurs also takes $O(n)$ time. Therefore, the algorithm as a whole takes $O(n)$ time.

(b)

Solution
Algorithm description
The following algorithm finds all the approximate medians of $a_1, \ldots, a_n$: 
1. Use the SELECT algorithm to find the \([n/4^{th}]\) element of the array. Call this element \(x\).
2. Use SELECT again to find the \([3n/4^{th}]\) element of the array. Call this element \(y\).
3. Iterate through the array and list each element \(a_i\) such that \(a_i \geq x\) and \(a_i \leq y\).

Correctness
If an element is greater than the \([n/4^{th}]\) element and less than the \([3n/4^{th}]\) element, then there are at least \(n/4\) elements less than the element and at least \(n/4\) elements greater than the element. If an element is less than the \([n/4^{th}]\) element, there are not \(n/4\) elements less than the element. If an element is greater than the \([3n/4^{th}]\) element, there are not \(n/4\) elements greater than the element.

Analysis
Each run of the SELECT algorithm can be done in \(O(n)\) time, and it takes \(O(n)\) time to iterate through the array to compare each element to \(x\) and \(y\). So the entire algorithm takes \(O(n)\) time.