CSc 220: Algorithms
Homework 3 Solutions

Problem 1: The law firm of Gennaro & Gennaro is looking for a new office in Manhattan. Because they often have to hand deliver important parcels to their clients, they are looking for a location that minimizes the sum of the distances between their new office and the clients’ locations. They have \( n \) clients \( C_1, \ldots, C_n \) where each client is represented by a point in the plane \( C_i = (x_i, y_i) \). Because this is Manhattan where we travel on a grid, the distance between the new office \( O \) and the clients’ locations. They have

\[
\delta = \sum_{i=1}^{n} d(O, C_i) = \sum_{i=1}^{n} |x - x_i| + |y - y_i|
\]

is minimized.

- Give an algorithm that computes the above location in \( O(n) \) time.
- A very smart intern at the firm notices that some clients receive parcels more often than others. He suggests to the partners that the right quantity to minimize is

\[
\delta_f = \sum_{i=1}^{n} f_i \cdot d(O, C_i)
\]

where \( f_i \) is the frequency with which client \( C_i \) receives parcels. In other words the \( f_i \)'s are real numbers such that \( 0 \leq f_i \leq 1 \) and \( \sum_{i=1}^{n} f_i = 1 \). Modify your previous algorithm to determine the location that minimizes the above quantity. Your algorithm should still work in \( O(n) \) time.

Solution:

- Following the hint, let us assume that \( n \) is odd and consider first the one-dimensional case in which points are on a line, not on the plane. Each client is a single coordinate \( C_i = x_i \) and the distance is \( |x - x_i| \). Let \( x_m \) be the median of the \( x_i \)'s. Because \( n \) is odd, there are exactly \( \ell = (n - 1)/2 \) elements which are smaller than \( x_m \) (and of course \( \ell \) elements which are larger than \( x_m \)).

We claim that the value \( \delta(x) = \sum_{i=1}^{n} |x - x_i| \) is minimized for \( x = x_m \). To prove this claim, we arbitrarily pair each element smaller than \( x_m \) with an element larger than \( x_m \). Let \( x_i \leq x_m \leq x_j \) be such a pair.

Assume now that \( x < x_m \) (the case \( x > x_m \) is dealt symmetrically). We have two cases.

Case 1: \( x_i \leq x < x_m \leq x_j \). In this case

\[
|x - x_i| + |x - x_j| = |x_m - x_i| + |x_m - x_j|
\]

since \( x_i \rightarrow x \rightarrow x_m \rightarrow x_j \)

Case 2: \( x \leq x_i \leq x_m \leq x_j \). In this case

\[
|x - x_i| + |x - x_j| = |x_m - x_i| + |x_m - x_j| + 2|x - x_i|
\]

since \( x \rightarrow x_i \rightarrow x_m \rightarrow x_j \)

In any case for any such pair \( x_i, x_j \) we have that

\[
|x - x_i| + |x - x_j| \geq |x_m - x_i| + |x_m - x_j|
\]

and therefore \( \delta(x) \geq \delta(x_m) \).

For the 2-dimensional case, we point out that to minimize a Manhattan distance it is sufficient to minimize each dimension independently so the new office location is \( O = (x, y) \) where \( x \) is the median of the \( x_i \)'s and \( y \) is the median of the \( y_i \)'s.
• Again following the hint we modify the notion of median. We say that \( x_m \) is the weighted median of the \( x_i \)'s if

\[
W_S = \sum_{i \in S} w_i < 1/2 \quad \text{and} \quad W_L = \sum_{j \in L} w_j \geq 1/2
\]

where \( S = \{ i : x_i < x_m \} \) is the set of indices \( i \) such that \( x_i < x_m \) and \( L = \{ j : x_j > x_m \} \) is the set of indices \( j \) such that \( x_j > x_m \). Note that \( W_S + W_L + w_m = 1 \) and since \( W_S < 1/2 \) we have that \( W_L + w_m > W_S \).

We claim that the value \( \delta(x) = \sum_{i=1}^n w_i|x-x_i| \) is minimized for \( x = x_m \). To prove this claim, consider what happens when we compute \( \delta(x) \) for \( x < x_m \) (the case \( x > x_m \) is dealt analogously). The new value \( x \) partitions \( S \) into two sets

\[
S_1 = \{ i : x_i \leq x < x_m \} \quad \text{and} \quad S_2 = \{ i : x < x_i < x_m \}
\]

Define \( W_{S1} \) and \( W_{S2} \) analogously and note that \( W_{S1} + W_{S2} = W_S \). Let \( \Delta = x_m - x > 0 \) and consider what happens to the difference \( |x_i - x| \) in four different cases

- \( i \in S_1 \)
  \[
  |x_i - x| = |x_m - x| - \Delta
  \]
- \( i \in S_2 \)
  \[
  |x_i - x| = |x_m - x| + 2|x_i - x|
  \]
- \( i = m \)
  \[
  |x_i - x| = |x_i - x_m| + \Delta
  \]
- \( i \in L \)
  \[
  |x_i - x| = |x_i - x_m| + \Delta
  \]

By using the above equalities we see that

\[
\delta(x) = \delta(x_m) - W_{S1}\Delta - W_{S2}\Delta + 2 \sum_{i \in S_2} w_i|x_i - x| + (W_L + w_m)\Delta
\]

i.e.

\[
\delta(x) = \delta(x_m) + (W_L + w_m - W_S)\Delta + 2 \sum_{i \in S_2} w_i|x_i - x| > \delta(x)
\]

The last inequality is due to the fact that \( W_L + w_m > W_S \) and \( Delta > 0 \) (note that the last term is non-negative since it's the sum of distances, but could be zero if \( S_2 \) is empty).

Again the two dimensional problem can be solved by finding the weighted median of the \( x \) and \( y \) coordinates independently.

We still have to describe a linear time algorithm to find the weighted median. This is done similarly as the traditional median. The algorithm \textsc{Weighted-Median} takes as input the array \( A \) with the \( n \) values, the array \( W \) with the \( n \) weights and two parameters \( \lambda_S, \lambda_L \) that identify the element we are looking for. In the case of the weighted median \( \lambda_S = \lambda_L = 1/2 \). When we call \textsc{Weighted-Median} on \( A, W, \lambda_S, \lambda_L \) the algorithm works as follows.

- Partition the \( n \) elements in groups of 5 and find the (traditional) median of each of them (\( O(n) \) time).
- Use recursion to find the "medians of medians" (\( T(n/5) \) time).
- Partition around this element to build the sets \( S \) and \( L \) as above. If \( W_S < \lambda_S \) and \( W_L \leq \lambda_L \) you are done. Otherwise recurse on the appropriate set by changing the weight parameters appropriately.
  * If \( W_S > 1/2 \) then recurse on \( S \) with parameters \( \lambda'_S = \lambda_S \) and \( \lambda'_L = \lambda_L - W_L \)
  * If \( W_L > 1/2 \) then recurse on \( L \) with parameters \( \lambda'_S = \lambda_S - W_S \) and \( \lambda'_L = \lambda_L \)
Note that $|L|, |S| > \frac{3n}{10}$ and therefore the second recursion is at most on $\frac{7n}{10}$ elements, giving rise to the same recurrence as the running time of the traditional median, i.e. $O(n)$ time.

**Problem 2:** As we will see later in the class, a binary search tree (BST) is a data structure organized as a binary tree where each node $x$ holds a value $key[x]$. The important property of a BST is that for any subtree of a BST rooted at $x$, we have that $key[x]$ is larger than the keys of all the nodes stored on the left subtree of $x$, and smaller than the keys of all the nodes stored on the right subtree of $x$. An interesting consequence of this is that by performing an "inorder" walk of the tree it is possible to output the keys stored in the tree in sorted order. An inorder walk of the tree rooted at $x$ is the following recursive procedure

$\text{INORDER}(x)$
- If $x \neq \text{NIL}$;
  - $\text{INORDER}($Left$[x])$;
  - Print $key[x]$;
  - $\text{INORDER}($Right$[x])$

Notice that INORDER makes no comparisons since it only reads the elements in the tree in a specific order.

Given $n$ elements can you build a BST containing the elements in $O(n)$ time? If yes show your algorithm. If no, explain why you think it is impossible.

**Solution:** The problem was poorly worded. If we consider arbitrary elements where the only way to sort them is by using comparisons, then the answer is no. Indeed if you could create a BST with $O(n)$ comparisons, then you would be able to sort the elements with only $O(n)$ comparisons, which we know is impossible.

However in the text of the problem I said $O(n)$ time which might have confused some of you. If the elements can be sorted in $O(n)$ time using say counting sort or radix sort, then one can then build a BST in $O(n)$ time as well after sorting the elements. Note that this requires elements of a specific type for which $O(n)$-time sorting algorithms exists.

**Problem 3:** On input an array $A$ of $n$ elements, each of which is an integer in $[0..n^4]$, describe a simple method for sorting $A$ in $O(n)$ time.

**Solution:** The idea is to write a number in $[0..n^4]$ in $n$-ary notation. In other words, if $0 \leq a \leq n^4$, then $a$ can be uniquely encoded as $(a_0, a_1, a_2, a_3)$ where $0 \leq a_i \leq n$, and $a = \sum a_i n^i$.

You can now use radix sort to sort the elements. You are going to do 4 passes where in each pass you are sorting $n$ numbers between 1 and $n$. By using a stable version of counting sort this will take $O(n)$ per pass so $O(n)$ total.