

CSE 3101: Solution for Assignment 2

Question 1 [13] Throughout this Question, the distance between two points on the two-dimensional plane is taken to be their Mahattan distance, i.e., the distance between two points $A = \langle x_1, y_1 \rangle$ and $B = \langle x_2, y_2 \rangle$ is

$$d(A, B) = |x_1 - x_2| + |y_1 - y_2|$$

- a) Give a divide-and-conquer algorithm that, given a set P of points represented by their x - and y -coordinates, outputs the smallest (Manhattan) distance of all pairs of points in the P . Your algorithm must run in time $n \log n$. You may assume that the points have distinct x -coordinates.
- b) Prove the correctness of your algorithm. (It suffices to show that the combining step in your algorithm is correct.)

Solution 1a [8]

As in the divide-and-conquer algorithm for the Closest Pair problem presented in class, first we sort the input into increasing order of x - and y -coordinates.

1. Sort P in increasing order of x -coordinates, Let X be the result.
2. Sort P in increasing order of y -coordinates, Let Y be the result.
3. Return Mahattan-Closest-Pair-Rec(P, X, Y)

The Mahattan-Closest-Pair-Rec program is similar to the Closest-Pair-Rec program.

Mahattan-Closest-Pair-Rec(P, X, Y):

1. If $|P| \leq 3$ then *brute-force*
2. Else
3. Compute line ℓ dividing P into P_L, P_R
4. Compute X_L, Y_L and X_R, Y_R
5. $\langle q_0, q_1 \rangle = \text{Mahattan-Closest-Pair-Rec}(P_L, X_L, Y_L)$
6. $\langle r_0, r_1 \rangle = \text{Mahattan-Closest-Pair-Rec}(P_R, X_R, Y_R)$
7. $\delta = \min\{d(q_0, q_1), d(r_0, r_1)\}$
8. $S =$ points in P within distance δ to ℓ *in increasing order of y -coordinates*
9. For each s in S ,
10. Compute its Mahattan distances to the next 12 points
11. Let $\langle s_0, s_1 \rangle$ be the closest pairs in S
12. End For
13. return the closest of $\langle q_0, q_1 \rangle, \langle r_0, r_1 \rangle$ and $\langle s_0, s_1 \rangle$
14. End If

Note that computing the line ℓ (in line 3) can be done in constant time, since the points in P is already sorted in increasing x -coordinate (the input X). Also, computing the set S (in line 8) takes linear time, since the points in P are already sorted in increasing y -coordinate. Similarly, computing X_L, X_R, Y_L, Y_R also takes linear time, using the input X and Y . As a result, the running time of Mahattan-Closest-Pair-Rec(P, X, Y) satisfies the recurrence:

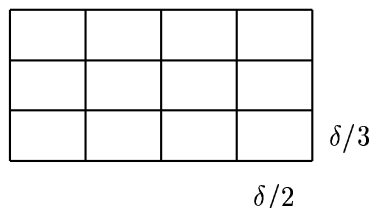
$$T(n) = 2T(n/2) + \mathcal{O}(n)$$

Therefore $T(n) = \mathcal{O}(n \log n)$.

Solution 1b [5]

For the combining step, any two points, one in P_L and one in P_R , that have Mahattan distance $\leq \delta$, must both be within distance δ from the line ℓ . This shows that for the combining step, it suffices to consider distances between pairs of points in the set S constructed in line 8.

Now for any two points in S , if their Mahattan is $\leq \delta$, then they must lie in a $\delta \times 2\delta$ rectangle:



Any two points in a $\delta/3 \times \delta/2$ rectangle have Mahattan distance at most $\delta/2 + \delta/3$, which is $< \delta$. So each $\delta/3 \times \delta/2$ rectangle contains at most one point from P_L and one point from P_R . Moreover, if such a rectangle contains exactly one point from P_L and one point from P_R , then one of these two points must be on the separating line ℓ . Since all points in P have distinct x -coordinate, it follows that all but at most one such rectangle contain at most one point from S . Consequently, it suffices to examine, for each point in S , its distances to the next 12 points in the ordering.

Question 2 [11] Consider the “Counting Significant Inversion Problem”. The input is an array of n natural numbers $A[1 \dots n]$. A “significant inversion” in this array is a pair $\langle i, j \rangle$ where $i < j$ and $A[i] > 2A[j]$. For example, in the following array

A	3	9	5	2
	1	2	3	4

there are 2 significant inversions: $\langle 2, 4 \rangle$ and $\langle 3, 4 \rangle$.

- a) Give a Divide-and-Conquer algorithm for counting the number of significant inversion. Your algorithm should run in time $\mathcal{O}(n \log(n))$.
- b) Verify that the running time of your algorithm is $\mathcal{O}(n \log(n))$.

Solution 2a [8]

As in the Counting Inversion Problem, here we also sort the input array at the same time as we count the number of significant inversions. Different from the algorithm for Counting Inversion, here we count the number of significant inversions separately (the subroutine Count-significant) before merging.

Significant-Inversion(A, p, r)

1. If $r \leq p$ return 0
2. Else
3. $q \leftarrow \lfloor \frac{p+r}{2} \rfloor$
4. $c_1 \leftarrow \text{Significant-Inversion}(A, p, q)$
5. $c_2 \leftarrow \text{Significant-Inversion}(A, q + 1, r)$
6. $c_3 \leftarrow \text{Count-significant}(A, p, q, r)$
7. Merge(A, p, q, r)
8. return $c_1 + c_2 + c_3$.
9. End If

The Merge procedure is presented in class. Here is the program Count-significant. Note that for indices i, j where $p \leq i \leq q$, and $q + 1 \leq j \leq r$, if $A[i] \leq 2A[j]$, then there are no significant inversions between $A[i]$ and any $A[k]$, where $j \leq k \leq r$. Also, if $A[i] > 2A[j]$, then $A[i] > 2A[k]$ for all $q + 1 \leq k \leq j$ (so $A[i]$ is responsible for $j - q$ significant inversions).

Count-significant(A, p, q, r)

1. $count \leftarrow 0, i \leftarrow q, j \leftarrow r$
2. While $i \geq p$ do
3. If $A[i] \leq 2A[j]$ then
4. If $j = q + 1$ return count
5. Else $j \leftarrow j - 1$
6. End If
7. Else
8. $count \leftarrow count + (j - q)$
9. If $i = p$ return count
10. Else $i \leftarrow i - 1$
11. End If
12. End While
13. End While

Solution 2b [3]

The Merge procedure takes linear time, and Count-significant also take linear time (we only go through the subarray $A[p \dots r]$ once). So the running time $T(n)$ of the algorithm satisfies

$$T(n) = 2T(n/2) + \Theta(n)$$

Therefore $T(n) = \Theta(n \log(n))$.

Question 3 [16] The Making Change Problem is the problem of, given a set of coin denominations (such as $\{1, 5, 10, 25, 100, 200\}$ for the Canadian currency), and a value n cents, finding the smallest number of coins needed that add up to n . For example, one way of producing 395 cents in Canadian coins is

1 tooney, 1 looney, 2 quarters, 4 dimes, 1 nickel

which requires 9 coins. A better way is

1 tooney, 1 looney, 3 quarters, 2 dimes

which requires only 7 coins.

In this question you are asked to design a greedy algorithm which, although may not work for all currencies, must work for the Canadian currency. Formally, the denominations are given in an array:

$$C[1], \dots, C[k]$$

(So for the Canadian currency, the input might be

$$C[1] = 25, C[2] = 10, C[3] = 1, C[4] = 200, C[5] = 100, C[6] = 5$$

Here $k = 6$.)

- a) Give a greedy algorithm which, given input the array C of denominations and a value n , output the number of coins of each type needed to add up to n .
- b) Show that your algorithm produces an optimal solution for the Canadian currency.
- c) What is the running time of your algorithm, as a function of k and *the length of the binary representation of n* ?

Solution 3a [8]

The idea is to use big coins first. So we will first sort the array of coins. Here is the algorithm:

1. Sort the array C so that

$$C[1] \leq C[2] \leq \dots \leq C[k]$$

2. $S \leftarrow \emptyset$ % the solution sequence of coins
3. While $n > 0$ do
4. Let j be the largest index so that $C[j] \leq n$
5. $S \leftarrow S \circ C[j]$
6. $n \leftarrow n - C[j]$
7. End While
8. Output S

Solution 3b [5]

We show that the algorithm produces an optimal solution for the Canadian coins.

Let S_i be the partial solution after loop i . We show that S_i can be extended to an optimal solution, for all i .

Base case: $i = 0$, $S_0 = \emptyset$. The claim is true.

Induction step: Assume that S_i can be extended to an optimal solution OPT , we show that S_{i+1} can also be extended to some optimal solution.

Let

$$OPT = S_i, b_{i+1}, \dots, b_\ell$$

where b_{i+1}, \dots, b_ℓ are in $\{1, 5, 10, 25, 100, 200\}$.

Without loss of generality, we can assume that

$$b_{i+1} \geq b_{i+2} \geq \dots \geq b_\ell$$

Let n_i be the value of n after loop i , then

$$n_i = b_{i+1} + \dots + b_\ell$$

In loop $i + 1$, the choice of j is such that $C[j] \leq n_i$, and if $j < k$ then $n_i < C[j + 1]$. We only have to show that $b_{i+1} = C[j]$.

In fact, since $b_{i+1} \leq n_i$, and $C[j]$ is the largest coin which is $\leq n_i$, we have $b_{i+1} \leq C[j]$. So assume by way of contradiction that $b_{i+1} < C[j]$. Notice that if $n_i = C[j]$, then S_{i+1} is already a solution, and therefore must be an optimal solution, and we are done.

So suppose that $C[j] < n_i$. Now we have

$$b_{i+1} < C[j] < n_i = b_{i+1} + \dots + b_\ell$$

so there must be the an (unique) index t , $g \geq i + 1$, so that

$$b_{i+1} + b_{i+2} + \dots + b_t < C[j] < b_{i+1} + b_{i+2} + \dots + b_{t+1} \quad (1)$$

($C[j]$ can not be of the form $b_{i+1} + b_{i+2} + \dots + b_t$ for any t : If this is the case, we can replace in OPT the whole block $b_{i+1}, b_{i+2}, \dots, b_t$ by $C[j]$ and obtain a better solution, which is impossible by the optimality of OPT.)

From (1) we have $b_{t+1} > 1$ (i.e., b_{t+1} is not the penny). Thus, using the hint, we know that in $b_{i+1}, b_{i+2}, \dots, b_t, b_{t+1}$ there are at most 1 loonie, 3 quarters, 2 dimes and 1 nickel.

We consider different cases for $C[j]$:

- $C[j] = 200$: the sum of 1 loonie, 3 quarters, 2 dimes and 1 nickel is exactly 200, so the second inequality in (1) cannot happen.
- $C[j] = 100$: then in $b_{i+1}, b_{i+2}, \dots, b_t, b_{t+1}$ there are at most 3 quarters, 2 dimes and 1 nickel. Again, the second inequality in (1) cannot happen.
- $C[j] = 25$: then in $b_{i+1}, b_{i+2}, \dots, b_t, b_{t+1}$ there are at most 2 dimes and 1 nickel. The second inequality in (1) also cannot happen.
- $C[j] = 10$: in $b_{i+1}, b_{i+2}, \dots, b_t, b_{t+1}$ there is just a nickel, and the second inequality in (1) cannot be satisfied.

In all the cases we reached contradiction. Thus it must be the case that $b_{i+1} = C[j]$. Hence S_{i+1} can be extended to an optimal solution. QED.

Solution 3c [3]

For a fixed currency (i.e. a fixed array C), the running time of the above algorithm is $\Theta(n)$. Note that sorting the array C takes time $\Theta(k \log(k))$.

Question 4 [10] Show that an array of n natural numbers, each is in the range $\{0, 1, 2, \dots, n^3 - 1\}$, can be sorted in time $\mathcal{O}(n)$. (Hint: A number in $\{0, 1, \dots, n^3 - 1\}$ can be expressed in the form

$$an^2 + bn + c$$

where $0 \leq a, b, c \leq n - 1$.)

Solution 4 [10]

Notice that the representation of numbers in $\{0, 1, \dots, n^3 - 1\}$ given in the hint provides a way of comparing two numbers: Suppose that $m, k \in \{1, 2, \dots, n^3 - 1\}$, where

$$\begin{aligned}m &= an^2 + bn + c \\k &= a'n^2 + b'n + c'\end{aligned}$$

then $m > k$ if and only if

$$\begin{cases} a > a' & \text{or} \\ a = a' \text{ and } b > b' & \text{or} \\ a = a' \text{ and } b = b' \text{ and } c > c' \end{cases}$$

In other words, we can view each number in $\{0, 1, \dots, n^3 - 1\}$ as a 3-digit number $\langle a, b, c \rangle$ where $0 \leq a, b, c < n$.

We can now use Radix sorting. Here the range of each digit is $k = n$, and there are 3 digits, so $d = 3$. The running time for Radix sorting is $\Theta(d(n + k)) = \Theta(n)$.