CS 170 Algorithms Fall 2013 Satish Rao HW 3

1. (11 pts.) Problem 2.5

a) $T(n) = 2T(n/3) + 1 = \Theta(n^{\log_3 2})$ by the Master theorem.

b) $T(n) = 5T(n/4) + n = \Theta(n^{\log_4 5})$ by the Master theorem.

c) $T(n) = 7T(n/7) + n = \Theta(n \log_7 n)$ by the Master theorem.

d) $T(n) = 9T(n/3) + n^2 = \Theta(n^2 \log_3 n)$ by the Master theorem.

e) $T(n) = 8T(n/2) + n^3 = \Theta(n^3 \log_2 n)$ by the Master theorem.

f) $T(n) = 49T(n/25) + n^{3/2} \log n = \Theta(n^{3/2} \log n)$. Apply the same reasoning of the proof of the Master Theorem. The contribution of level i of the recursion is

$$\left(\frac{49}{25^{3/2}}\right)^i n^{3/2} \log\left(\frac{n}{25^i}\right) = \left(\frac{49}{125}\right)^i O(n^{3/2} \log n)$$

Because the corresponding geometric series is dominated by the contribution of the first level, we obtain $T(n) = O(n^{3/2} \log n)$. But, $T(n)$ is clearly $\Omega(n^{3/2} \log n)$. Hence, $T(n) = \Theta(n^{3/2} \log n)$.

g) $T(n) = T(n-1) + 2 = \Theta(n)$.
First, we expand the tree. Each level takes 2 units of time, and we notice that our subproblem decreases by size one each time, so we have $\Theta(n)$ levels. Thus, the overall running time is $\Theta(n)$.

h) $T(n) = T(n-1) + n^c = \sum_{i=1}^{n} i^c + T(0) = \Theta(n^{c+1})$.
This is very similar to the last part, which is where we get the sum (the sum starts from n, and goes down to 1, but we notice that we can just write it in the sigma notation above). Notice that this is also a result you have already proved in HW1.

i) $T(n) = T(n-1) + c^n = \sum_{i=1}^{n} c^i + T(0) = \frac{c^{n+1}-1}{c-1} + T(0) = \Theta(c^n)$.
Notice that we get the sum in a very similar way to the previous part. However, since the sum is an increasing geometric series, we know the last term dominates, and thus is in $\Theta(c^n)$ (rewriting it in the form $\frac{c^{n+1}-1}{c-1} = \Theta(c^n)$ is another way to see this as well).

j) $T(n) = 2T(n-1) + 1 = \sum_{i=0}^{n-1} 2^i + 2^n T(0) = \Theta(2^n)$.

k) $T(n) = T(\sqrt{n}) + 1 = \sum_{i=0}^{k} 1 + T(b)$, where $k \in Z$ such that $n^{\frac{1}{2^k}}$ is a small constant b, i.e. the size of the base case. This implies $k = \Theta(\log \log n)$ and $T(n) = \Theta(\log \log n)$.

2. (15 pts.) Problem 2.23

a) If A has a majority element v, v must also be a majority element of A_1 or A_2 or both. To find v, recursively compute the majority elements, if any, of A_1 and A_2 and check whether one of these is a majority element of A. The running time is given by $T(n) = 2T(\frac{n}{2}) + O(n) = O(n \log n)$.

b) We assume n is even. If not, pick an element and check if it is a majority element. If not, discard the element. If a majority existed among n elements, it will be a majority of $n-1$ elements.

Let B be the array remaining after the procedure, and let k be the number of elements of B. Suppose A has a majority element v but v is not a majority of B. Then the number of elements which are not v in B is at least $k/2$. These correspond to $\geq k$ elements in A which got paired up with themselves. Since there are at most $k/2$ elements of v in B and at least $n/2$ elements of v in A, there are $\geq n/2 - k$ elements in A which are not v but got paired up with v. Since these elements got paired up with v, they are distinct from the $\geq k$ elements which got paired up with themselves. However, there are $> n/2$ elements of v in A, so A must have $> n/2 + (n/2 - k) + k = n$ elements, a contradiction. Thus, if A has a majority element, it is also a majority of B. We did not prove if and only if, so if a majority of B exists, we still need to check to see if it is a majority of A.

The running time of this algorithm is described by the recursion $T(n) = T(n/2) + O(n)$. Hence, $T(n) = O(n)$.

3. (20 pts.) Problem 2.32

a) Suppose 5 or more points in L are found in a square of size $d \times d$. Divide the square into 4 smaller squares of size $\frac{d}{2} \times \frac{d}{2}$. At least one pair of points must fall within the same smaller square: these two points will then be at distance at most $\frac{d}{\sqrt{2}} < d$, which contradicts the assumption that every pair of points in L is at distance at least d.

b) The proof is by induction on the number of points. The algorithm is trivially correct for two points, so we may turn to the inductive step. Suppose we have n points and let (p_s, p_t) be the closest pair. There are three cases.
If $p_s, p_t \in L$, then $(p_s, p_t) = (p_L, q_L)$ by the inductive hypothesis and all the other pairs tested by the algorithm are at a larger distance apart, so the algorithm will correctly output (p_s, p_t). The same reasoning holds if $p_s, p_t \in R$.
If $p_s \in L$ and $p_t \in R$, the algorithm will be correct as long as it tests the distance between p_s and p_t. Because p_s and p_t are at distance smaller than d, they will belong to the strip of points with x-coordinate in $[x-d, x+d]$. Suppose that $y_s \leq y_t$. A symmetric construction applies in the other case. Consider the rectangle S with vertices $(x-d, y_s), (x-d, y_s+d), (x+d, y_s+d), (x+d, y_s)$. Notice that both p_s and p_t must be contained in S. Moreover, the intersection of S with L is a square of size $d \times d$, which, by a), can contain at most 4 points, including p_s. Similarly, the intersection of S with R can also contain at most 4 points, including p_t. Because the algorithm checks the distance between p_s and the 7 points following p_s in the y-sorted list of points in the middle strip, it will check the distance between p_s and all the points of S. In particular, it will check the distance between p_s and p_t, as required for the correctness of the algorithm.

c) When called on input of n points this algorithm first computes the median x value in $O(n)$ and then splits the list of points into those belonging to L and R, which also takes time $O(n)$. Alternatively, the algorithm can assume the points are sorted by x coordinate (which will require an upfront cost of $O(n\log n)$ to sort the original list). Then the algorithm can recurse on these two subproblems, each over $n/2$ points. Once these have been solved the algorithm sorts the points in the middle strip by y coordinate, which takes time $O(n\log n)$ and then computes $O(n)$ distances, each of which can be calculated in constant time. Hence the running time is given by the recursion $T(n) = 2T(\frac{n}{2}) + O(n\log n)$. This can be analyzed as in the proof of the Master theorem. The kth level of the recursion tree will contribute $t_k = 2^k \frac{n}{2^k}(\log n - k)$. Hence, the total running time will be:

$$\sum_{k=0}^{\log n} t_k = n\log^2 n - n\sum_{k=0}^{\log n} k \leq n\log^2 n - \frac{n}{2}\log^2 n = O(n\log^2 n).$$

d) We can save some time by sorting the points by y-coordinate only once up front and making sure that the split routine is implemented as not to modify the order by y when splitting by x. Sorting takes time $O(n\log n)$, while the time required by the remaining of the algorithm is now described by the recurrence $T(n) = 2T(\frac{n}{2}) + O(n)$, which yields $T(n) = O(n\log n)$. Hence, the overall running time is $O(n\log n)$. Alternatively, the sorting by y-coordinate can be done working up the recursion tree, by merging sorted lists from L and R. Another approach is to use bucket sort on the middle strip, using $d \times d$ squares as buckets.

4. (15 pts.) Practice with polynomials and complex numbers

(a) $\omega^7 = e^{2\pi i(7/8)} = \frac{\sqrt{2}}{2} - \frac{\sqrt{2}}{2}i = \omega^{-1}$, so $\omega + \omega^7 = \sqrt{2}$

(b) $p(1) = 2$, $p(\omega) = 1+i$, $p(\omega^2) = 0$, $p(\omega^3) = 1-i$

(c) $q(x) = 1 + x + x^3$

Here's a way to use Matlab's `vander` command to compute the vector of coefficients:

```
b = [3; 1+sqrt(2)*1i; 1; 1+sqrt(2)*1i];
w = exp(2*1i*pi/8);
A = vander([1 w w^2 w^3]);
A\b
```

5. (20 pts.) Finding the missing integer We show how we can reduce this problem to one of size $N/2$ in linear time. Consider the least significant bits in A, and count the number of 1 bits found. If N was even, we expect $N/2$ 1 bits found if all integers were present, otherwise, we expect $(N+1)/2$ 1 bits found. Let p be the actual count, and q be the expected count. We then have two cases:

- If $p = q$, we know the least significant digit of the missing integer must be 0.
- If $p = q - 1$, we know the least significant digit of the missing integer must be 1.

Now, suppose c was the least significant digit that we found. We can then ignore all positions in A where there was a $1 - c$ as the least significant digit (this can be done by creating an auxiliary

array B containing all the interesting indices we are considering in A, which can be created in $O(N)$ time if we are ignoring bit complexity $O(N \log N)$ time otherwise). Note, we must copy indices, not the actual values in the array, since we don't want to actually look at the numbers in there yet. Notice if we ignore those positions, and the least significant digits, we get the problem of finding the missing integer in an array of length at most $N/2$. Thus, we can recursively apply our algorithm to get a number h, and then return $2h+c$ as our result ($2h$ to shift h by one, and then add back in the least significant digit).
Notice, counting the number of 1 bits takes $O(N)$ time. This reduces the problem size in half, so we have the recurrence $T(N) = T(N/2) + O(N)$ which counts the number of bits, which, by master's theorem, gives us $T(N) = O(N)$. The copying indices over will take $O(N \log N)$ time ($O(N)$ is fine here if you forgot to include bit complexity).

6. **(20 pts.) Pareto points** First, sort the points in increasing x coordinate, breaking ties with increasing y coordinates. (so in the example, we would have $\{(1,1),(2,4),(3,3),(4,2)\}$). We now give two different solutions (only one of these is necessary):

   - BY DIVIDE AND CONQUER (slower, but ok): First, if there are any two points with same x coordinate, we can discard all points except for the one with biggest y coordinate (since those points can not be Pareto points). We can then assume all x coordinates are distinct from here on out. Split the points up, and recursively find the Pareto points, and suppose they were returned in sorted x coordinate. Now, we show how to construct the Pareto points of the S (and also in sorted x coordinate). Let L be the left set, and R be the right set, and let P_L be the pareto points of the left set and P_R be the Pareto points of the right set. Now, we can make 3 important observations:
     - Any Pareto point in S must be in one of the sets P_L, P_R (or in other words $P_S \subseteq P_L \cup P_R$). This is clearly true, since any Pareto point in S must satisfy the condition on any subset of points in S, which would imply it was in P_L or P_R.
     - All Pareto points in R are also Pareto points in S (in other words, $P_R \subseteq P_S$)
       Each Pareto point identified in R satisfies the condition on all the points in R. Since each point in R has x-coordinate strictly bigger than any point in L, each Pareto point in R will also satisfy the condition on all points in L, so is a Pareto point of the entire set S.
     - For a Pareto point in L to be a Pareto point in S, it must have y coordinate strictly bigger than the first y coordinate of a Pareto point in R.
       Each Pareto point in L satisfies the condition on all points in L, so in order for it to be a Pareto point in S, it must satisfy the condition on all points in R. But since the x-coordinate of points in L is smaller, in order to satisfy the condition, it must have y coordinate strictly larger than any y coordinate in R. Additionally, the largest y coordinate in R must also be a Pareto point (if there are ties, you can take the largest x), and additionally, must have the smallest x. Suppose not, then there is a Pareto point (x', y'), with $x' < x, y' < y$, which is a contradiction (this can't be a Pareto point), so since the points are in sorted-by-x order, the first Pareto point in R must have the largest y coordinate.

     Using these three observations, we can see how to do a linear sweep to merge P_R and P_L. For any point in P_L, we compare its y coordinate with the first y coordinate in P_R, and keep it only if it is strictly larger. Then, we can add all points in P_R. We can see this keeps the points in sorted order, since all the x-coordinates in P_L are strictly smaller than x coordinate in

P_R, thus, we have successfully merged our smaller sub-problems. The recurrence in this case is $T(n) = 2T(n/2) + O(n)$, which simplifies to $O(n \log n)$ time. Thus, since sorting takes $O(n \log n)$, and finding the Pareto points from this list takes $O(n \log n)$, the overall running time is $O(n \log n)$.

- BY ONE SWEEP (faster, not necessary for full credit): Let y_{max} be the maximum y coordinate found so far, and initially set it to $-\infty$. Now, iterate through the points in backwards order, and keep a set P of Pareto optimal points that we have found. Whenever a point's y coordinate exceeds y_{max}, we put that point into P, and then update y_{max} to be what we found. We now claim that this algorithm is correct. To show this, we need to show that all Pareto optimal points must be in P and P only contains Pareto points.
  - Let $p = (x, y)$ be an arbitrary point, and p is a Pareto point. We show that this implies the algorithm will put $p \in P$. We know $\forall (x', y') \in S, x' < x$ or $y' < y$. Since we have sorted the points in order, and we are iterating backwards, we know that any point with $x' < x$ must be processed after p, so we only need to consider the case when $x' \geq x$. However, since p is a Pareto point, and $x' \geq x$, we must have $y' < y$. Thus, y is bigger than any previously processed y, so is bigger than y_{max}, so then p will be added into P by our algorithm, as expected.
  - Now, suppose $p \in P$. We now show that p must be a Pareto point. Since we sorted the points, any point processed after p will either have $x' < x$ or $x' = x \wedge y' < y$. Additionally, the condition for getting into P requires that y is bigger than y_{max}, thus, we must have $y' < y$ for points processed before p. Thus p is a Pareto point as wanted.

  Since membership in P is an if and only if relationship, we can conclude that P is exactly the set we want. As for run time, we sort the points which takes $O(n \log n)$ comparisons, each of which takes $O(1)$ time. We do a linear scan afterwards, which takes $O(n)$ time. Thus the total running time is $O(n \log n)$.