1: Recurrences

(a) $T(n) = 4T(n/4) + n$.

The depth of the tree: $m = \log_4 n$

k^{th} layer in the recursion tree of this relation:

$$= 4^k \cdot T(n/4^k) + 4^{k-1} \cdot \frac{n}{4^{k-1}}$$
$$= 4^k \cdot T(n/4^k) + n$$

where, $k \in \{1, m\}$

$T(n)$ can be written in terms of $T(1)$ (bottom-most layer) as follows:

$$= 4^m \cdot T(1) + \sum_{i=1}^{m} n$$
$$= n \cdot T(1) + n \cdot \sum_{i=1}^{m}(1)$$
$$= n \cdot T(1) + n \cdot m$$
$$= n \cdot T(1) + n \cdot \log_4 n$$
$$\implies O(n) + O(n \log n)$$
$$\implies O(n \log n)$$

(b) $T(n) = 4T(n/4) + 1$.

The depth of the tree: $m = \log_4 n$

k^{th} layer in the recursion tree of this relation:

$$= 4^k \cdot T(\tfrac{n}{4^k}) + 4^{k-1} \cdot 1$$

where, $k \in \{1, m\}$

$T(n)$ can be written in terms of $T(1)$ (bottom-most layer) as follows:

$$= 4^m \cdot T(1) + \sum_{i=1}^{m} 4^{i-1} \cdot 1$$
$$= n \cdot T(1) + (\sum_{i=1}^{m} 4^i)/4$$
$$= n \cdot T(1) + (4^m - 1)/12$$
$$= n \cdot T(1) + (n - 1)/12$$
$$\implies O(n)$$

(c) $T(n) = T(n-1) + n$.

The depth of the tree: $m = n - 1$

k^{th} layer in the recursion tree of this relation:

$$= T(n-k) + n - k + 1$$

where, $k \in \{1, m\}$

$T(n)$ can be written in terms of $T(1)$ (bottom-most layer) as follows:

$$= T(1) + (n + (n-1) + (n-2) + \cdots + 3 + 2)$$
$$= T(1) + (n + (n-1) + (n-2) + \cdots + 3 + 2 + 1) - 1$$
$$= T(1) + n(n+1)/2 - 1$$
$$\implies O(n^2)$$

(d) $T(n) = T(n/3) + T(n/2) + \sqrt{n}$.

At each step there are some reproducing (T()) and some non-reproducing ($n^{1/2}$). We will consider them separately.

The reproducing terms at the k^{th} layer can be written as,

$$\binom{k}{0} \cdot T(n/(3^k \cdot 2^0)) + \binom{k}{1} \cdot T(n/(3^{k-1} \cdot 2^1)) + \cdots + \binom{k}{k} \cdot T(n/(3^0 \cdot 2^k))$$

From recursion tree we can identify a the Pascal tree kind of structure and hence the above expression. Out of all terms in the above expression, $T(n/2^k)$ is the one decreasing by least amount in each step. Therefore, the depth, m, of the tree can be considered to be dictated by this term and is $\log_2 n$

At the bottom-most layer the reproducing terms will amount to,

$$= T(1)(\binom{m}{0} + \binom{m}{1} + \cdots + \binom{m}{m}) \qquad = T(1)(2^m)$$
$$= T(1) \cdot n$$

The non-reproducing terms at the k^{th} layer can be written as,

$$\sqrt{n} \cdot \{\binom{k-1}{0}/(3^{(k-1)/2} \cdot 2^{0/2})) + \binom{k-1}{1} \cdot /(3^{(k-1)/2} \cdot 2^{1/2}) + \cdots + \binom{k-1}{k-1} \cdot /(3^{0/2} \cdot 2^{(k-1)/2})\}$$

Using the binomial expansion, this expression can be written as,

$$\sqrt{n}(\frac{1}{\sqrt{3}} + \frac{1}{\sqrt{2}})^{k-1} = 1.284^{k-1}$$

At the bottom-most layer the non-reproducing terms will amount to,

$$\sqrt{n} \cdot \sum_{i=1}^{m} 1.284^{i-1}$$
$$= c \cdot \sqrt{n} \cdot (1.284^m - 1)$$
$$\implies O(n^{0.86})$$

Therefore, the overall bound can be written as,

$$\implies O(n) + O(n^{0.86})$$
$$\implies O(n)$$

(e) $T(n) = T(\sqrt{n}) + 4$.

Let the depth of the tree be m

k^{th} layer in the recursion tree of this relation:

$$= T(n^{1/2^k}) + 4$$

where, $k \in \{1, m\}$

Let's say that the recursion bottoms out when $n^{1/2^k}$ becomes a constant c. Calculating the value of m,

$$n^{1/2^m} = c$$
$$(1/2^m) \cdot \log n = \log c$$
$$\log_c n = 2^m$$
$$m = \log_2 \log_c n$$

$T(n)$ can be written in terms of $T(c)$ (bottom-most layer) as follows:

$$= T(c) + \sum_{i=1}^{m} 4$$
$$= T(c) + 4 \cdot m$$
$$= T(c) + 4 \cdot \log_2 \log_c n$$
$$\implies O(\log(\log n))$$

(f) $T(n) = 3T(n/2) + g(n)$.

The depth of the tree: $m = \log_2 n$

k^{th} layer in the recursion tree of this relation:

$$= 3^k \cdot T(n/2^k) + 3^{k-1} \cdot g(n/2^{k-1})$$

where, $k \in \{1, m\}$

$T(n)$ can be written in terms of $T(1)$ (bottom-most layer) as follows:

$= 3^m \cdot T(1) + \sum_{i=1}^{m} 3^{i-1} \cdot g(n/2^{i-1})$
$= n^{log_2 3} \cdot T(1) + \sum_{i=1}^{m} 3^{i-1} \cdot g(n/2^{i-1})$
$= n^{1.58} \cdot T(1) + \sum_{i=1}^{m} 3^{i-1} \cdot g(n/2^{i-1})$

We will use the following expression for the evaluation of closed form for different values of $g(n)$

$$n^{1.58} \cdot T(1) + \sum_{i=1}^{m} 3^{i-1} \cdot g(n/2^{i-1}) \tag{1}$$

i) $g(n) = n^2$

Using Eqn 1,

$= n^{1.58} \cdot T(1) + \sum_{i=1}^{m} 3^{i-1} \cdot (n/2^{i-1})^2$
$= n^{1.58} \cdot T(1) + n^2 \cdot \sum_{i=1}^{m} (3/4)^{i-1}$
$= n^{1.58} \cdot T(1) + n^2 \cdot 4/3 \cdot \sum_{i=1}^{m} (3/4)^{i}$
$= n^{1.58} \cdot T(1) + 4n^2 \cdot (1 - (3/4)^m)$
$= n^{1.58} \cdot T(1) + 4n^2 - 4n^{1.58}$
$\implies O(n^2)$

ii) $g(n) = n$

Using Eqn 1,

$= n^{1.58} \cdot T(1) + \sum_{i=1}^{m} 3^{i-1} \cdot (n/2^{i-1})$
$= n^{1.58} \cdot T(1) + n \cdot \sum_{i=1}^{m} (3/2)^{i-1}$
$= n^{1.58} \cdot T(1) + 2n \cdot ((3/2)^m - 1)$
$= n^{1.58} \cdot T(1) + 2n^{1.58} - 2n$
$\implies O(n^{1.58})$

iii) $g(n) = n^{log_2 3}$

Using Eqn 1,

$= n^{1.58} \cdot T(1) + \sum_{i=1}^{m} 3^{i-1} \cdot (n/2^{i-1})^{log_2 3}$
$= n^{1.58} \cdot T(1) + \sum_{i=1}^{m} 3^{i-1} \cdot (n^{log_2 3})/3^{i-1}$
$= n^{1.58} \cdot T(1) + \sum_{i=1}^{m} n^{log_2 3}$
$= n^{1.58} \cdot T(1) + n^{log_2 3} \cdot \sum_{i=1}^{m} (1)$
$= n^{1.58} \cdot T(1) + n^{log_2 3} \cdot \log_2 n$
$= n^{1.58} \cdot T(1) + n^{1.58} \cdot \log_2 n$
$\implies O(n^{1.58} \log n)$

2: Sorting "nearby" numbers

Algorithm: We're given the array $A[0, \ldots, N-1]$ and the difference, M, of the largest and smallest element of A. Create an array of size M called C (with initial elements all zero), that will contain the number of occurences of values starting from $min_i(A[i])$ to $max_i(A[i])$ in A. The first element in C will denote the no. of occurences of $min_i(A[i])$, second element will denote the occurence of $min_i(A[i]) + 1$ and so on until the last element, which will denote the no. of occurences of $min_i(A[i])$. Now scan through the array A and increment the value of $C[A[i] - min_i(A[i])]$ by 1. This process will take $O(N)$ time.

Create an array of size = size(A) called S, that will be sorted version of A. Next scan through the array C and write-out index of C (plus $min_i(A[i])$) in S if the current entry in C is non-zero. If the current entry in C is 1, then write this index (plus $min_i(A[i])$) in S; if the current entry in C is 2, then write this index (plus $min_i(A[i])$) in the next two places in S and so on. At the end of these iterartions S will contain the sorted version of A. This second set of iterations will take $O(M)$ time.

Correctness: The size of the array C is sufficient to contain the occurences of all the elements of A. Each index of C denotes the no. of occurences of the element of A that is equal to this index. Therefore, when looping over these indices (for the array C) in the ascending order in the second part of the algortihm, we are making sure that the elements in S are always written in the ascending order

Running time: The first part of the algorithm takes $O(N)$ time and the second part of the algorithm takes $O(M)$ time. Therefore, the total running time of the algorithm is $O(N + M)$.

3: Selecting in a Union

Algorithm: The pseudocode for the algorithm is shown in Algorithm 1. The basic idea behind the algorithm is that we're reducing the search space and changing the value of k accordingly. The recursion bottoms out when the size of one of the array becomes zero and we return the k^{th} element of the other array.

Algorithm 1 Finding the k^{th} smallest element in Union of arrays A and B

```
1: procedure k-smallest($A[0 : N-1]$,$B[0 : N-1]$,k)
2:     if $len(A) == 0$ then
3:         return B[k]
4:     else
5:         return A[k]
6:     end if
7:     $centA = len(A)/2$
8:     $centB = len(B)/2$
9:     if $k > (centA + centB)$ then
10:        if $A[centA] > B[centB]$ then
11:            return k-smallest($A[:]$, $B[(centB + 1):]$, $k - centB - 1$)
12:        else
13:            return k-smallest($A[(centA + 1):]$, $B[:]$, $k - centA - 1$)
14:        end if
15:    else
16:        if $A[centA] > B[centB]$ then
17:            return k-smallest($A[:(centA)]$, $B[:]$, $k$)
18:        else
19:            return k-smallest($A[:]$, $B[:(centB)]$, $k$)
20:        end if
21:    end if
22: end procedure
```

The lines 2-6 of the algorithm are the recursion bottom-out clauses described above. Next we denote the center-most element of A and B by $centA$ and $centB$, respectively. In line 9, we see if the value of k is on the right-hand side or the left-hand side of the union of A and B. If it's on the right hand side (If part of line 9), we again perform a check if the center-most element of A is larger than the corresponding element of B. If it is larger, this implies that $centA$ is larger than the first half of B (as the arrays are sorted) and since we are guaranteed that k^{th} element lies somewhere beyond the sum of $centA$ and $centB$, we only need to search in second-half of B and whole A. The function is called again with these new arrays and the value of k changed to $k - centB - 1$.

The remaining part of the algorithm is just the complement of the logic discussed until now with the value of k modified accordingly.

Correctness: The bottom-out cases relies on the fact that the two arrays are sorted because we return the k^{th} element of the non-empty array. The algorithm only splits the arrays without changing the sorted order of any one of them. The conditionals statements reduce the size of the search space depending on the value of k.

If k is greater than the sum $centA + centB$, then the k^{th} element is beyond the middle elements of either array. By comparing the middle values of both the arrays we can safely discard the half portion of one of the arrays. The value of k is reduced by the size of the half portion of the split array.

For the other case, similar comparisons are made but the value of k is kept unchanged. Since we are splitting only one of the arrays in a recursive step, only one of the arrays will get empty in the last recursion. At this recursion, we will get the value of k from the non-empty array.

Running time: The recurrence relation of the algorithm can be written as,

$$T(n) = T(3n/4) + c$$

where, n is the sum of size of the two arrays ($n = 2N$), and c is some constant.

One of the arrays become half at each layer of the tree, and therefore, this relation.

The depth of the tree: $m = \log_{4/3} n$

k^{th} layer in the recursion tree of this relation:

$$= T((3/4)^k n) + \text{c}$$

where, $k \in \{1, m\}$

$T(n)$ can be written in terms of $T(1)$ (bottom-most layer) as follows:

$$= T(1) + \sum_{i=1}^{m} c$$
$$= T(1) + c \cdot m = T(1) + c \cdot \log_{4/3} n$$
$$\implies O(log(n))$$

4: Closest pair of restaurants in Manhattan

(a) Let's say there are four points, $A : (1, 2), B : (2, 4), C : (3, 3)$, and $D : (5, 6)$ in the 2D plane. The first step of the algorithm can find $x = 2$ so that $L = \{(1, 2), (2, 4)\}$ and $R = \{(3, 3), (5, 6)\}$. The only pair of restuarant in L is (A, B) and the distance between them is 3. Similarly, the only pair of restuarant in R is (C, D) and the distance between them is 5. The step 2 of the algorithm will return $d = min(3, 5) = 3$. But there exist a pair of points (B, C) with distance 2. Therefore, without the steps 3 and 4, the algorithm produces a wrong output.

(b) Let's assume that there's a square of size $d \times d$. Divide the square into six smaller boxes of equalr area $= d^2/6$. Let's say the dimension of the boxes is $d/2 \times d/3$. The maximum distance between any two points in this box can be $d/2 + d/3 = 5d/6$, which is less than d. So we cannot fit more than one point in a box such that the distance between them is greater than or equal to d. Therefore, if the square contains more than 6 (one per each smaller box) points, then the distance between two of them is strictly smaller than d.

The steps 1 and 2 of the algorithms find a pair of points that have the smallest distance d in the two halves, L and R. The steps 3 and 4, then try to find a pair such that one of them lie in L and the other in R and have a distance, $d' < d$, between them. The halves L and R are divided by the line x. Consider the vertical strip bounded by the lines, $x - d$ to the left and $x + d$ to the right. Also consider some horizontal line y and the line $y + d$. The area bounded by these lines contains two squares, S_L and S_R of size $d \times d$ on either side of x (S_L in L and S_R in R).

The steps 1 and 2 guarantee that the minimum distance between any two points in L and R is d. Combining this with the result proved above, we can say that S_L and S_R can atmost have 6 points each. Let's assume, without loss of generality, that there's a point, P, in the combined region of S_L and S_R, such that it lies on the line y. Any point that lies above the line $y + d$, will be a distance, greater than d, apart from P. Therefore, we only need to compute the distance of P from the remaining atmost 11 consecutive points (in increasing order of their y-coordinate) to determine if there's another point Q in this region with the distance $(P \rightarrow Q) < d$. This arguments remains valid when we relax the number of such computations to 13.

(c) We can sort the set of given n points based on their x-coordinates which will add $O(n \log n)$ to the overall complexity. We can keep these sorted points in an array. Everytime this array or its subset is passed as an argument to the recursive step, the value of x can be found by accesing the middle element of the input array.

If during a recursive call, the algorithm finds that there are only two points in the input array, it will return (to the previous step) the distance between them without doing any further splitting. The previous step will receive two distances from L and R halves respectively. It will compare these two values to find the distance, d, at this step. Next the steps 3 and 4 will find if there is a pair with distance $d' < d$ in the merged space of L and R. The minimum of all these distances is returned to its previous step. At each step, the algorithm makes two recursive calls (on the partition L and R) with half no. of points at the current step. Also steps 3 and 4 perform sorting on the current set of points (based on their y-coordinate). Therefore, the recurrence relation for this algorithm can be written as,

$$T(n) = 2T(n/2) + O(n \log n)$$

The depth of the tree: $m = \log_2 n$

k^{th} layer in the recursion tree of this relation:

$$= 2^k \cdot T(n/2^k) + 2^{k-1} \cdot (n/2^{k-1})(\log(n/2^{k-1}))$$
$$= 2^k \cdot T(n/2^k) + n \cdot (\log(n/2^{k-1}))$$

where, $k \in \{1, m\}$

$T(n)$ can be written in terms of $T(1)$ (bottom-most layer) as follows:

$$2^m \cdot T(1) + \sum_{i=1}^{m}(n)(\log(n/2^{i-1}))$$
$$n \cdot T(1) + n \cdot \sum_{i=1}^{m}(\log(n) - \log(2^{i-1}))$$
$$n \cdot T(1) + n \cdot \sum_{i=1}^{m}(\log(n) - (i-1)\log(2))$$

$$n \cdot T(1) + n \cdot \sum_{i=1}^{m}(\log(n) - i\log 2 + \log(2))$$
$$n \cdot T(1) + n \cdot (m(\log n + \log 2) - \log 2 \sum_{i=1}^{m} i)$$
$$n \cdot T(1) + n \cdot (m(\log n + \log 2) - \log 2 \cdot (m(m+1)/2)$$
$$n \cdot T(1) + n \cdot (const \cdot \log^2 n)$$
$$\implies O(n\log^2 n)$$

5: Linear Time Median

(a) The first step of the algorithm is to divide arbitrarily, the array A into groups of 5 which is a constant time operation. The sorting of each fixed sized array $(B_1, B_2, \ldots, B_{n/5})$ is again a constant time operation. For $n/5$ such arrays, it will take $O(n)$ time. We will get the 3^{rd} (middle element) element of each of these $n/5$ arrays to form C. This again will take $O(n)$ time. The time for finding the median of an n sized array is $T_{median}(n)$. Since the size of C is $n/5$, the time for finding its median is $T_{median}(n/5)$. Therefore, the total time for this algorithm is $T_{median}(n/5) + O(n)$.

(b) Let's consider a permutation, π, on the arrays $B_1, B_2, \ldots B_{n/5}$ such that they are arranged in the ascending order of their middle elements i.e. in the permutation, if $B_i[3] < B_j[3]$, then B_i is placed before B_j. Also, let's denote the first array in this order by X_1^π, the second by X_2^π and the last one by $X_{n/5}^\pi$.

M is the middle-most element of the C and in the permutation described above it must be the 3^{rd} of the array $X_{n/10}^\pi$, which is $X_{n/10}^\pi[3]$. Since, $X_1^\pi[3] \le X_2^\pi[3] \le \cdots \le X_{n/10}^\pi[3]$, we can say that $X_{n/10}^\pi[3]$ is greater than or equal to the first three elements of the arrays, $X_1^\pi, X_2^\pi, \ldots, X_{n/10}^\pi$. The total number of such elements is $3n/10$. Therefore, we can say that $M = X_{n/10}^\pi[3]$ is greater than or equal to atleast $3n/10$ elements of the array A. Because $3n/10 > n/4$, M is greater than or equal to atleast $n/4$ elements in A.

Similarly, as $X_{n/10}^\pi[3] \le X_{n/10+1}^\pi[3] \le \cdots \le X_{n/5}^\pi[3]$, we can say that $X_{n/10}^\pi[3]$ is smaller than or equal to last three elements of the arrays, $X_{n/10+1}^\pi, X_{n/10+2}^\pi, \ldots, X_{n/5}^\pi$. The total number of such elements is again $3n/10$. Therefore, M is smaller than or equal to $3n/10$ elements of A or M is smaller than or equal to atleast $n/4$ elements of A.