1: Recurrences

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

There will be k levels of recursion where $k = log_4(n)$. The amount of work done at each level k is summed. There is $T(1) * n$ work done by the bottom level.

$$\sum_{i=1}^{k} 4^{i-1} * \frac{n}{4^{i-1}} + T(1) * n$$

$$\sum_{i=1}^{k} n + T(1) * n$$

$$k * n + T(1) * n$$

$$log_4(n) * n + c * n$$

$\boxed{O(n \log n)}$

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

$$\sum_{i=1}^{k} 4^{i-1} * \frac{1}{4^{i-1}} + T(1) * n$$

$$\sum_{i=1}^{k} 1 + T(1) * n$$

$$k + T(1) * n$$

$$log_4(n) + c * n$$

$\boxed{O(n)}$

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

There will be k levels of recursion where $k = n - 1$.

$$\sum_{i=1}^{k} n - 1 + \sum_{i=1}^{k} n$$

$$\sum_{i=1}^{n-1} n - \sum_{i=1}^{n-1} 1 + \sum_{i=1}^{n-1} n$$

$$\frac{n(n-1)}{2} + (n-1) + \frac{n(n-1)}{2}$$

Here I use the fact that the sum of the first $n-1$ numbers is equal to $\frac{n(n-1)}{2}$.

$\boxed{O(n^2)}$

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

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

(fa) $T(n) = 3T(n/2) + n^2$

(fb) $T(n) = 3T(n/2) + n$

(fc) $T(n) = 3T(n/2) + n^{log_2 3}$

2: Sorting "nearby" numbers

Algorithm: : The sorting algorithm I am going to describe is a modified version of counting sort, that will accommodate negative integers and a range of numbers not beginning at zero. Let C[0...M-1] be an array that will store the counts of the integers in A, all initially starting at 0.

Iterate A and for each $element_A$ in A, index $C[element_A - minA]$ and add one to the count.

Iterate C and update each count to the first index that an element with a value of $index_C$ from A will be placed in the output array (the index of C represents the value of an element from A). This can be done by summing all the counts in C with all counts that precede each count in C.

Final iteration of A to create the sorted output array $O[1...n-1]$. We will index C with each element from A to find the index to place the element in O, subtracting off the offset of $minA$ that accommodates negative integers. Additionally, we will increase the count in C for the current element.

We are done and $O[1...n-1]$ will contain the elements of $A[1...n-1]$ in sorted order.

Correctness: :

Running time: : This sorting algorithm performs three iterations. The first iteration of A scans n elements performing constant work with each one. Next C is scanned and a "rolling sum" is applied to M elements, which is constant work. A final iteration of A is performed over n items to construct the output array. Giving $c * n + c * M + c * n$ work or $O(n + M)$.

$\boxed{O(n + M)}$

3: Selecting in a union

Algorithm 1 Union Select

```
1: procedure unionSelect(Al, Ah, Bl, Bh, A, B, k)
2:     if Ah < Al then
3:         return B[k - Al]
4:     end if
5:     if Bh < Bl then
6:         return A[k - Bl]
7:     end if

8:     Am = (Al + Ah)/2
9:     Bm = (Bl + Bh)/2

10:    if B[Bm] >= A[Am] then
11:        if k <= Am + Bm then
12:            return unionselect(Al, Ah, Bl, Bm - 1, A, B, k)
13:        else
14:            return unionselect(Am + 1, Ah, Bl, Bh, A, B, k)
15:        end if
16:    else
17:        if k <= Am + Bm then
18:            return unionselect(Al, Am - 1, Bl, Bh, A, B, k)
19:        else
20:            return unionselect(Al, Ah, Bm + 1, Bh, A, B, k)
21:        end if
22:    end if
23: end procedure
```

The goal of the algorithm is to discard half of either A[0...n-1] or B[0...n-1] each recursion. There are four choices to discard (bottom-half or top-half of either A or B). The arrays are sorted, so looking at the middle value of each array will give us information about the values that preside and follow it.

Case 1: The middle element of B is greater than the middle element of A and k is less than the sum of the two middle indices. Since $B[Bm]$ is greater than $A[Am]$, we know that all the elements before index Am come before the element at index Bm (if all the elements where in one sorted array). Since k is less than $Am + Bm$, we know k is one of these elements or could be in A. This allows us to discard the top half of B.

Case 2 - 4: The other three cases follow the same logic as case 1: determine a half of either A or B that the kth smallest element couldn't exist in and discard it.

The recursion continues until one of the arrays becomes empty. At which point the other array is indexed and the kth smallest element is returned.

Correctness: :

Running time: : In the worst case, both arrays will be divided in half until one array is empty and the other only contains one element. This can be viewed as performing binary search on each array, which has a worst-case complexity of $O(\log n)$. There will be two "binary searches", one on A and one on B giving $O(\log n) + O(\log n)$ which simplified is $O(\log n)$.

A recurrence relation can also be used to prove the runtime. Let $N = n + n$. There is one recursion per level, 3/4ths of the remaining N is passed into each call, and there is constant work done at each level.

$$T(N) = T(3N/4) + 1$$

Which when solved with master theorem gives $O(\log n)$.

$O(\log n)$

4: Closest pair of restaurants in Manhattan

5: Linear Time Median