1: Recurrences

Collaboration with Alessandro Ferrero and Maks Cegielski-Johnson

(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 summation of the first $n-1$ numbers is equal to $\frac{n(n-1)}{2}$.

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

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

Using the master theorem we can analyze $T(n) = 2T(n/3) + \sqrt{n}$ and $T(n) = 2T(n/2) + \sqrt{n}$ to find bounds on the original recurrence relation.

$$T(n) = 2T(n/3) + \sqrt{n} = O(n^{log_2 3})$$
$$T(n) = 2T(n/2) + \sqrt{n} = O(n)$$

This gives the bounds:

$$O(n^{log_2 3}) \leq T(n) = T(n/3) + T(n/2) + \sqrt{n} \leq O(n)$$

$\boxed{O(n^{log_2 3}) \leq T(n) \leq O(n)}$

(e) $T(n) = T(\sqrt{n}) + 4$ Let n = 2^m.

$$T(2^m) = T(2^{m/2}) + 4$$
$$T(x) = T(x/2) + 4$$

Using the masters theorem, we get $O(logX)$. Substituting back for n, gives $O(log(log(n)))$.

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

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

There will be k levels of recursion where $k = log_2(n)$.

$$\sum_{i=1}^{k} 3^{i-1}(\frac{n}{2^{i-1}})^2 + T(1) * 3^k$$

$$\sum_{i=1}^{k} 3^{i-1}\frac{n^2}{2^{(i-1)*2}} + T(1) * 3^k$$

$$n^2 \sum_{i=1}^{k} \frac{3^i * 3^{-1}}{4^i * 2^{-2}} + T(1) * 3^k$$

$$\frac{4n^2}{3} \sum_{i=1}^{k} \frac{3^i}{4^i} + T(1) * 3^k$$

The summation is a geometric series and converges to 3 as k approaches infinity. k is equal to $log_2(n)$ so it won't converge, but can be viewed as a constant approximately equal to 3.

$$c * n^2 + T(1) * 3^{log_2(n)}$$

$$c * n^2 + c * n^{log_2(3)}$$

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

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

There will be k levels of recursion where $k = log_2(n)$.

$$\sum_{i=1}^{k} 3^{i-1} \frac{n}{2^{i-1}} + T(1) * 3^k$$

$$n \sum_{i=1}^{k} \frac{3^{i-1}}{2^{i-1}} + T(1) * 3^k$$

$$\frac{2n}{3} \sum_{i=1}^{k} \frac{3^i}{2^i} + T(1) * 3^k$$

$$\frac{2n}{3} \sum_{i=0}^{k} (\frac{3}{2})^i + T(1) * 3^k$$

The summation is a geometric series, substitute in the partial sum formula:

$$\frac{2n}{3} * \frac{\frac{3}{2}(\frac{3}{2}^k - 1)}{\frac{3}{2} - 1} + T(1) * 3^k$$

Combining the constant terms into c_1 and c_2 and focus on the n terms:

$$c_1 * n^{log_2(3)} + c_2 * 3^{log_2(n)}$$

Using the laws of log, these terms are the same.

$\boxed{O(n^{log_2(3)})}$

(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: : We are given $M = max_i A[i] - min_i A[i]$. Given the range of numbers that can appear in A, we are safe to store in C the number of elements that will appear before a given element. With the sums in C, we can iterate C and populate the output array.

The indices of C are the values put into the output array and the indices of an array are always in ascending order.

The counts in C increases by 1 for each n elements, so the total counts in C will be equal to n. This ensures that the output array will have n elements and contain the same number of elements as A.

Therefore we have correctness as the numbers in the output array will be in ascending order and there will be n elements.

Running time: : This sorting algorithm performs three iterations (assuming M is given to us, if not M can be determined with an additional iteration through A). 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

Collaboration with Maks Cegielski-Johnson

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: :

We don't know if the *kth* smallest element exists in A[1..n-1] or B[1..n-1], so our beginning search space is of size of $N = |A + B|$. Each recursive call discards N/4 of the search space. This continues until the high and low index of one of the array crosses each other. Then the other array (the one whose indexes didn't cross over) can be index by k offset by the low index of the other array. Since both A and B are sorted, this will return the *kth* smallest index in the union of the two arrays.

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

(a) Consider the set of points {(k,0), (k+3, 0), (k+5,0), (k+8,0)}.

Step 1: We will find $x = k + 4$, such that $\frac{n}{2}$ points have $x_i \leq x$. This will partition the points into L = {(k,0), (k+3,0)} and R {(k+5,0), (k+8,0)}.

Step 2: We will find the smallest distance in A to be 3 and the smallest distance in B to be 3. We will return $d = 3$ as the minimum distance.

This algorithm is incorrect as the distance between points (k+3, 0) and (k+5, 0) is 2.

(b)

Correctness: : Given 7 points $p_1, ..., p_7$ in a d x d square, where d is the minimum distance between the n points, there must be two points that have a distance strictly smaller than d.

The best case scenario is when there are 5 points located at:
$(0, 0),\ (0, d),\ (d, 0),\ (d, d)$ *and* $(d/2, d/2)$. Or simply put the 4 corners and the center of the square (if d is odd, there are 4 centers and any will do for this proof). It's easy to see that this is the optimal set of points as adjacent corners have a distance of d and all the corners have a distance of d from the center (assuming d is even and there is one center point).

Using the pigeonhole principle, an additional point can't be added to the square without causing two points to have a distance strictly smaller than d.

(c)

Algorithm: :

Step 1: Let P be the set of all points. Sort the elements (with merge sort) of P based off their x-coordinate, we'll call this P'. Let x be equal to middle element P'.

Step 2: Call the function recursively twice: once on L and once on R. Let d be equal to the minimum of the two returned distances.

Step 3: Calculate $[x - d, x + d]$ based on x from step 1 and d from step 2. Iterate through P', placing the points with an x-coordinate in range into a new list S.

Step 4: Sort S (with merge sort) based on their y-coordinate, forming S'. For each point, compute the distance between it and the 13 points ahead of it in the list. Let d' be equal to the minimum point found this way.

Step 5: Return both d and d'.

Running time: :

Step 1: Sorting n elements will take $O(nlogn)$ time.

Step 2: The algorithm is called recursively two times, passing in half of the current n each time.

Step 3: Iteration through P and performing constant work on each element will take at most $O(n)$ time.

Step 4: Sorting a ratio of n elements will take $O(nlogn)$ time. Additionally, S will be searched a constant number of times for each element which will take $O(n)$ work.

Overall the algorithm will take $O(nlogn)$ time on each level of recursion. There will be two recursive calls while $n > 2$ and they each will work with n/2 elements. This gives the recurrence relation:

$$T(n) = 2T(n/2) + O(nlogn)$$

Proving that it leads to the bound $T(n) \leq O(nlog^2 n)$:

$$\sum_{i=1}^{k} 2^{i-1} \frac{nlog_2(n)}{2^{i-1}} + T(1) * 2^k$$

Where k is the number of levels of recursion which will be $log_2(n)$ levels:

$$\sum_{i=1}^{k} nlog_2(n) + T(2) * 2^k$$

$$log_2(n) * nlog_2(n) + T(2) * n$$

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

5: Linear Time Median

(a) To prove that $T_{near-median}(n) \leq T_{median}(n/5) + O(n)$ we will look at at the work done at each step of the procedure.

Step 1: To copy A into a total of n/5 arrays, each of length 5, will require O(n) work. Each element in A will have to be touched once.

Step 2: The B arrays of length 5 now must be sorted and there are n/5 of them. Each sort will take $5log_2(5)$ work (assuming merge sort) which is a constant amount of work. Giving $c * (n/5)$ or $O(n)$ complexity.

Step 3: To form C, the middle item of each array of size 5 is taken an placed in C. This is constant work and must be done for n/5 arrays. Giving $c * (n/5)$ or $O(n)$ complexity.

Step 4: To find M the median of the numbers in C, we will call $T_{median}(C)$. C contains one element from each of the arrays of size 5, so it's size is n/5. Giving $T_{median}(n/5)$.

The total work is found by summing the four steps, giving us the expected runtime.

$$T_{median}(n/5) + O(n)$$

(b) In step 3 of the algorithm, we take the middle element of each array of size 5 to form C. We now know there are at least two elements larger and smaller than the middle element. We will have n/5 of these elements.

In step 4, we let M be the median of the numbers in C. We now know there are 2n/5 elements (which is greater than n/4) in C larger and smaller than M. This proves there exists at least n/4 elements in A that are $\leq$ M and there are at least n/4 elements of A that are $\geq$ M.