CSE 21 Fall 2022
Homework 3 sample problems

Students should feel free to discuss these problems.
KEY CONCEPTS Recursive algorithms, recurrence relations.

1. Solve the following recurrence relations to give closed form expressions of the functions:

   (a) $A[0] = 1$, $A[n] = 2A[n-1] + 1$ for $n \geq 1$.
   $A[n] = 2A[n-1]+1 = 2(2A[n-2]+1)+1 = 4A[n-2]+2+1 = 4(2A[n-3]+1)+2+1 = 8A[n-3]+4+2+1 = 2^kA[n-k]+2^{k-1}+2^{k-2}+...1 = 2^kA[n-k]+(2^k-1)$ (The last step is a sum we simplified in a previous assignment, so we can just use it.) Using this for $k = n$, $A[n] = 2^nA[0] + 2^n - 1 = 2^n + 2^n - 1 = 2^{n+1} - 1$. Alternatively, we could guess $A[n] = 2^{n+1} - 1$ by working through a few examples, and prove it by induction: $A[0] = 1 = 2^1 - 1$, and if $A[n] = 2^{n+1} - 1$, then $A[n+1] = 2A[n]+1 = 2(2^{n+1}-1)+1 = 2^{n+2}-2+1 = 2^{n+2}+1$.

   (b) $B[0] = 1$, $B[n] = B[n-1] + 2n + 1$ for $n \geq 1$
   Working through a few examples, $B[1] = 1+2+1 = 4$, $B[2] = 4+4+1 = 9$, $B[3] = 9+6+1 = 16$, and we conjecture that $B[n] = (n+1)^2$. We can prove this conjecture by induction. $B[0] = 1 = (0+1)^2$ for the base case, and if $B[n] = (n+1)^2$, $B[n+1] = (n+1)^2+2(n+1)+1 = ((n+1)+1)^2$ as desired.

   (c) $C[0] = C[1] = 1$, $C[n] = 2C[n-2]$ for $n \geq 2$.
   Unravelling, $C[n] = 2C[n-2] = 4C[n-4] = 8C[n-6] = ...2^kC[n-2k]$. We reach one of the two base cases when $n-2k = 0$ or $n-2k = 1$, i.e., when $k = \lfloor n/2 \rfloor$. At this point, if n is even, $C[n] = 2^{\lfloor n/2 \rfloor}C[0] = 2^{\lfloor n/2 \rfloor}$ and if n is odd, $C[n] = 2^{\lfloor n/2 \rfloor}C[1] = 2^{\lfloor n/2 \rfloor}$ In either case, $C[n] = 2^{\lfloor n/2 \rfloor}$

   (d) $D[0] = 1, D[n] = nD[n-1]$ for $n \geq 1$.
   Unravelling, $D[n] = nD[n-1] = n(n-1)D[n-2] = .. = n(n-1)..(n-k)D[n-k-1]$. We reach the base case when $k = n-1$, so $D[n] = n(n-1)....1 = n!$.

   (e) $E[0] = E[1] = 1, E[n] = 2E[\lfloor n/2 \rfloor]$ for $n \geq 2$. Unravelling, $E[n] = 2E[\lfloor n/2 \rfloor] = 4E[\lfloor n/4 \rfloor] = ..2^kE[\lfloor n/2^k \rfloor]$. We stop when $\lfloor n/2^k \rfloor$ reaches the base case 1, i.e., when $k = \lfloor \log_2 n \rfloor$. At this k, $E[n] = 2^kE[1] = 2^{\lfloor \log_2 n \rfloor}$.

2. Give a recursive description of selection sort. You can use a sub-routine $ArgMinimum(A[1..n])$ which returns the location of a minimum value in array $A[1..n]$ and takes linear time $O(n)$.

   Prove this recursive algorithm correctly sorts the input from smallest to largest. Give a recurrence relation for its time , and solve the recurrence relation to give the time up to order.

The idea of Min Sort is to find the location of the minimum element, swap it with the first element, and repeat on the remainder of the array.

We can express this recursively as

$RMinSort(A[1..n])$

(a) IF $n = 1$ return A.

(b) $M = ArgMinimum(A[1..n])$.

(c) Swap $A[M]$ and $A[1]$

(d) $RMinSort(A[2..n])$.

We prove this correctly sorts by induction on the array size n. If $n = 1$, then the array A is already sorted, and returning A correctly returns a sorted list with the same elements.

Assume $RMinSort$ correctly sorts lists with $n-1$ elements. On $A[1..n]$, by the guarantee for $ArgMinimum$, for $M = ArgMinimum(A[1..n])$, $A[M]$ is the smallest value in the array. Once we swap, $A[M]$ is in the first position, and all other elements are in the other positions. By the induction hypothesis, the recursive call correctly sorts the other positions. So the final output is the smallest element, followed by a sorted list of all other elements, in other words, all elements in sorted order.

Let $T(n)$ be the worst-case time for $RMinSort$ on an array of size n. We are told $ArgMinimum$ takes $\Theta(n)$ time for some n. All of the other steps except the recursive call take constant time. The recursive call takes at most $T(n-1)$ time. Thus, $T(n) \leq T(n-1) + cn$ for some constant c. $T(1)$ is some constant time c'. Unravelling, $T(n) \leq T(n-1) + cn \leq T(n-2) + c(n-2) + c(n-1) \leq T(n-3) + c(n-3) + c(n-2) + c(n-1) \leq ...T(n-k) + c(n-k) + c(n-k+1)....cn$. Using this when $k = n-1$, we get $T(n) \leq T(1) + c(1) + c2 + ...cn \leq cn^2 + c' \in O(n^2)$. (I could get an exact expression using "Gauss's sum", but we don't need an exact expression, so I'm just using that we have n terms all at most cn.)

3. Describe how to use MergeSort to solve the Element Distinctness problem. How much time total does this take, in order notation?

   After a list is sorted, if there are duplicates, they will be in consecutive positions. So we can sort the list using any method, then see if $A[I] = A[I+1]$ for any $1 \leq I \leq n-1$. This last step takes time $O(n)$. We saw that MergeSort takes time $O(n \log n)$, so the total time would be $O(n \log n + n) = O(n \log n)$.

4. Design a recursive algorithm for the following problem: The input is an array of integers $A[1..n]$ so that $A[1] > A[n]$. The problem is to find an integer I so that $1 \leq I \leq n-1$ and $A[I] > A[I+1]$, i.e., a particular place where the list is not sorted. Prove that your algorithm is correct, and give a time analysis up to orderusing a recurrence relation.

This is a recursive version of the algorithm from last assignment. When we make a recursive call to a subarray, we can just pass the new end markers for the subarray, rather than copy the array. Technically, we should make the array itself a global variable and have the inputs to the recursive procedure just be the end markers, but I think the following is clearer:

$LocalUnsorted(A[I..J]$:array of integers with $A[I] > A[J]$.

```
(a) If $J = I + 1$ return $I$.
(b) $M = \lfloor (I + J)/2 \rfloor$
(c) IF $A[M] > A[M + 1]$ return $M$
(d) IF $A[I] > A[M]$ return $LocalUnsorted(A[I..M])$
(e) Return $LocalUnsorted(A[M + 1, , J])$.
```

We claim that this procedure returns a position K with $A[K] > A[K+1]$ whenever $A[I..J]$ is a $n = J - I + 1 \geq 2$ element array with $A[I] > A[J]$. We prove this by strong induction on $n \geq 2$. If $n = 2$, $J = I + 1$, and the precondition on the array is that $A[I] > A[J] = A[I + 1]$. Since the algorithm returns $K = I$, this is correct. I'm also going to handle $n = 3$ as a base case, because in this case $M + 1 = J$, making it a bit different. If $n = 3$, $J = I + 2$ and we set $M = I + 1$. Since $A[I] > A[J]$, if $A[M] \leq A[M + 1] = A[J]$, then $A[I] > A[M]$, so either $A[M] > A[M + 1]$ and we correctly return M, or $A[I] > A[M] = A[I + 1]$ and we return $LocalUnsorted(A[I, I + 1]) = I$ which is correct.

Assume, for some $n > 3$, that the algorithm is correct for each array $A[I..J]$ with $2 \leq J - I + 1 < n$ and $A[J] > A[I]$. Let $A[I..J]$ be such an array with $J-I+1 = n$. Then for $M = \lfloor (I+J)/2 floor$, $2 \leq M-I+1 < n$ and $2 \leq J - (M + 1) + 1 < n$ (here we need $n > 3$). If $A[M] > A[M + 1]$, we correctly return $K = M$. If not and $A[I] > A[M]$, $A[I..M]$ is an array of size at least 2 and less than n with the first value greater than the last, so by the induction hypothesis, the recursive call returns a position K with $A[K] > A[K + 1]$ as required. Finally, if neither $A[M] > A[M + 1]$ or $A[I] > A[M]$, then $A[M + 1] \geq A[M] \geq A[I] > A[J]$, so $A[M + 1..J]$ is an array with between 2 and $n - 1$ elements with the first value greater than the last. So by the induction hypothesis, in this case, the recursive call returns a position K with $A[K] > A[K + 1]$ as required.

So in all cases, the algorithm returns a position K with $A[K] > A[K + 1]$. By induction on n, this holds for all n.

The time for all steps except the recursive call is constant. The recursive call is always to an array of at most half the size of the orginal. Thus, for $T(n)$ the worst-case time the algorithm takes on inputs of size n, for some c, $T(n) \leq T(n/2) + c$. Then $T(n) \leq T(n/4) + c + c \leq T(n/8) + c + c + c...T(n/2^k) + ck$. We reach the base case, $n = 2$, when $k > log_2 n - 1$, at which point $T(n) < T(1) + c(log_2 n) \in O(\log n)$. Thus, the total time is $O(\log n)$.

5. Consider the following recursive algorithm to take a binary integer $b_{n-1}..b_0$ representing the integer $\sum_{i=0}^{i=n-1} b_i 2^i$ to decimal. It uses a sub-procedure *Add* that adds two decimal numbers.

   Convert$[b_{n-1}..b_0]$

   (a) If $n = 1$ return b_0.
   (b) $X = Convert(b_{n-1}..b_1)$
   (c) $Y = Add(X, X)$
   (d) $ReturnAdd(Y, b_0)$

   Prove that this algorithm returns the integer whose binary representation is the input. Give a time analysis using recurrence relations assuming *Add* is constnt time. Then give a time analysis assuming that *Add* takes linear time in terms of the number of decimal digits in the two inputs. (You will first have to think about how many digits is the decimal representation on an integer that takes n bits in binary.)

   We want to show that $Convert(b_{n-1}..b_0)$ returns the integer whose binary representation is $b_{n-1}, , b_0$, i.e., $\sum_{i=0}^{i=n-1} b_i 2^i$. We prove this by induction on n. For $n = 1$, we have $Convert(b_0) = b_0 = \sum_{i=0}^{i=0} b_i 2^i$, since $2^0 = 1$. Assume for any $n-1$ bit string $Convert(b'_{n-2}..b'_0) = \sum_{i=0}^{i=n-2} b'_i 2^i$. Then on input $b_{n-1}..b_0$, we first assign $X = Convert(b_{n-1}...b_1)$ By the induction hypothesis, $X = \sum_{i=0}^{n-2} b_{i+1} 2^i$, since in this input , $b'_i = b_{i+1}$. Then we assign $Y = 2X = 2\sum_{i=0}^{n-2} b_{i+1} 2^i = \sum_{i=0}^{n-2} b_{i+1} 2^{i+1} \sum_{i=1}^{n-1} b_i 2^i$ Finally, we add b_0, to get $(\sum_{i=1}^{n-1} b_i 2^i) + b_0 = (\sum_{i=0}^{n-1} b_i 2^i)$ the correct answer.

   For the time analysis assuming *Add* is constant time (i.e, numbers are small enough that we can perform an *Add* on hardware), then all but the recursive calls are now constant time. Then we get the recurrence $T(n) = T(n-1) + c$ for some c. This gives $T(n) \in O(n)$ since we've seen this recurrence many times already.

   If instead *Add* takes time proportional to the number of digits in the decimal representations, we need to think about how large these representations are. If z is the value of the number , the number of bits is at least $n \geq \log_2 z$ and the number of digits is at most $log_1 0z + 1 = (log_2 z)/(log_2 10) + 1 \leq n/log_2 10 + 1$. So the number of digits is growing at most linearly with the number of bits, and in fact is between a factor of 3 and of 4 smaller. Thus, the time to do the two *Adds* in this case will also be $O(n)$. Then we get $T(n) \leq T(n-1) + cn$ for some c, which we solved above to see that $T(n) \in O(n^2)$.