80977816 2/8/17
CS124 Homework 2

1. To prove the correctness of StoogeSort, I wll first show that StoogeSort will correctly sort lists of length 1, 2, and 3. The argument behind why StoogeSort correctly sorts a list of length 3 will generalize to a list of arbitrary length. If StoogeSort is passed a list of length 1, then it should just return taht list becuase it is trivially sorted. If StoogeSort is passed a list of length 2, check if the two elements are sorted. If they're not swap them and then return the sorted list. Now for a list of length 3, we look at the first 2/3 of the list, in this case the first two elements and sort them. After the first stage we know that whatever number is in the second position of the list is greater than whatever number is in the first position. Now we move onto the second stage and sort the last two thirds of the list. After the second stage is compete we know that the number in the second position is less than the number in the third position and that the number in the first position of the list is less than the number in the third position. Therefore the number in position 3 is in the correct spot because it is larger than the other two numbers, but there is no relationship (after the second stage) between the numbers in the first and second positions. Now we run the third stage (sort the first two thirds of the list again). After the third stage whatever number is in the first position must be less than the number in the second position and so because we already know that after stage 2, the first and second elements are less than the third element, the list must now be correctly sorted.

   Now that we have the base cases of 1,2,3 determined I will use an inductive argument to show that a list of size n can be sorted correctly using StoogeSort. Assume for the induction hypothesis that all lists smaller than n can be sorted correctly by StoogeSort. If n is not a multiple of 3, then the size of the list to be sorted in each stage of StoogeSort should be equal to $\lceil \frac{2}{3}n \rceil$ so the stages of StoogeSort always sort at least two thirds of the list and no elements are missed. So if the list had length 8, then the first stage would sort the first 6 elements and the second would sort the last 6 and so on. Now to prove that StoogeSort can sort a list correctly partition the list into three sections such that the length of section 1 + the length of section 2 is $\lceil \frac{2}{3}n \rceil$ and that the length of section 2 + the length of section 3 is also $\lceil \frac{2}{3}n \rceil$. In the first stage of StoogeSort, we sort sections 1 and 2, which can be done by the induction hypothesis because the length of these two sections is less than n. After stage 1, every number in section 1 is less than every number in section 2. On to stage 2. After stage 2 is completed, we know that every number in section 2 is less than in section 3 and because of the first stage we know that every number in section 1 is less than every number in section 3. As a result, the numbers in section 3 must be in their correct positions. In the last stage we sort sections 1 and 2 and so after stage 3, every number in section 1 is less than every number that comes after it in section 1 and less than every number in sections 2 and 3, and every number in section 2 correctly sorted within section 2 and is less than all the numbers in section 3. Thus StoogeSort sorts correctly.

   $T(n) = 3 \cdot T(\frac{2}{3}n)$ and $T(1) = 1$, which by the Master Theorem means that $T(n) = O(n^{log_{1.5}(3)}) \approx O(n^{2.7})$

2. (a) $T(1) = 1, T(n) = T(n-1) + 4n - 4$
   I will prove by induction that $T(n) = 1 + 2n(n-1)$
   Base case: $T(1) = 1 + 2 \cdot 1(1-1) = 1$ and so the base case holds. Induction hyptothesis: assume that $T(n) = 1 + 2n(n-1)$
   Consider $T(n+1)$. By the recurrence relationship, we know that $T(n+1) = T(n) + 4(n+1) - 4 = T(n) + 4n$. Now assuming the induction hypothesis, we have $T(n+1) = 1 + 2n(n-1) + 4n = 1 + 2n^2 + 2n = 1 + 2(n+1)n$ which is exactly the same form as the induction hypothesis and so the induction holds.

   (b) $T(1) = 1, T(n) = 2T(n-1) + 2n - 1$
   I will prove by induction that $T(n) = 3 \cdot 2^n - 2n - 3$
   Base case: $T(1) = 3 \cdot 2^1 - 2 - 3 = 6 - 5 = 1$ and so the base case holds.
   Induction hypothesis: assume that $T(n) = 3 \cdot 2^n - 2n - 3$
   Consider $T(n+1)$. By the recurrence, we have that $T(n+1) = 2T(n) + 2(n+1) - 1 = 2T(n) + 2n + 1$.

Now assume the induction hypothesis is true. Then $2T(n)+2n+1 = 2(3\cdot 2^n - 2n - 3) + 2n + 1 = 3\cdot 2^{n+1} - 4n - 6 + 2n + 1 = 3\cdot 2^{n+1} - 2n - 2 - 3 = 3\cdot 2^{n+1} - 2(n+1) - 3$, and so the induction holds.

(c) $T(n) = 4T(n/2) + n^3$
By the Master Theorem, $4 < 2^3$, which means that $T(n) = \Theta(n^3)$

(d) $T(n) = 17T(n/4) + n^2$
By the Master Theorem, $17 > 4^2$, which means that $T(n) = \Theta(n^{log_4(17)})$

(e) $T(n) = 9T(n/3) + n^2$
By the Master Theorem, $9 = 3^2$, which means that $T(n) = \Theta(n^2 logn)$

(f) $T(n) = T(\sqrt{n}) + 1$ Let $n = 2^{2^x}$. Therefore $T(n) = T(2^{2^x}) = T((2^{2^x})^{\frac{1}{2}}) + 1 = T(2^{2^{x-1}}) + 1$. This recurrence will continue until $x = 0$ and so $T(n) = T(2^{2^x}) = x$. Because $n = 2^{2^x}$, $x = log(logn)$. So, $T(n) = \Theta(log(logn))$.

3. (a) Let H be a binary min-heap that holds k elements and as a result take $O(log(k))$ time to insert or change an item in the heap and has constant time access to the minimum member of the heap which is stored at the top of the heap. To merge k sorted lists with n elements in total into one large list do the following. First, add the first element from each of the k lists to the heap, storing the value, index of the element, and the list that value belonged to in the heap. This takes $O(klogk)$ time so far. Now, so long as the heap isn't empty, pop off the minimum element of the heap (which is the smallest element that we haven't already put into the merged list of length n) and add it to our new merged array of length n and then check which of the k lists this element came from and if there is an element that came after this element in its list, insert that into the heap. If the element that was just removed from the heap was the last element in its original list, just continue. As a result, the heap never has more than k numbers stored in it. This process will take $O((n-k)logk)$ time because after inserting the first k elements into the heap there are still $n-k$ other elements. So this whole sorting process takes $O(klogk + (n-k)logk) = O(nlogk)$ time. By using a min heap, we ensure that the element that gets put into the list of length n is always the smallest one that hasn't already been put in the list.

   (b) Because we are trying to sort a list that is k-close to being sorted then if we take any interval of k elements and insert them into a binary min-heap, then the smallest element in that interval will be inserted at the top of the heap. Here's the algorithm to sort a k-close list. First, insert the first k elements into the min-heap H. If there are fewer than k elements in the list, insert the entire list. Because the list is k-close, the smallest element in the list must be among the first k elements. Once k elements have been inserted into the list, a process which takes $O(klogk)$ time, remove the element at the top and place it in the first position of the new sorted list of length n. Add the $k+1$ element of the list to the heap, if there is one, and then remove the new minimum of the list and place it in the second sport of the new sorted list. Continue this process of removing the minimum element and adding the next element in the unsorted list until the heap is empty. This will ultimately involved $O(nlogk)$ operations because it requires n insertions into the heap, each taking $logk$ time and a constant amount of work to insert it into the new list.

4. First perform a depth first search and rank the vertices according to their postorder number (topological sort) - this takes time $O(|V| + |E|)$ and keep track of every vertex's postorder number. Create an array d such that $d[i]$ will keep track of the length of the longest path to vertex i. Initialize all values of d to $-\infty$. Also create an array $prev$ such that $prev[i]$ will say what vertex came before i in the path. Initialize all the values of $prev$ to null. Now go through the vertices in their topological order. For every vertex v, look at all its neighbors u and if $d[v] + w(v,u) > d[u]$, where $w(v,u)$ is the weight along edge (v,u), then set $d[u]$ to $d[v] + w(v,u)$ and set $prev[u] = v$. Once this loop is done find the vertex with the longest path from a source by searching through d and then you can trace back through the path by looking at $prev$. In pseudocode this looks like:
LongestPath(DAG):

order = TopologicalSort(DAG)
$d = [-\infty, -\infty..., -\infty]$ (d has size $|V|$)
$prev$ = [null, null, ..., null] (prev has size $|V|$)
for v in order:
for all u adjacent to v:
if $d[u] < d[v] + w(v, u)$:
$d[u] = d[v] + w(v, u)$
$prev[u] = v$
$i, p = max(d)$ (max does a linear search returns the length of the longest path, p ending in vertex i)
ptr = prev[i]
while(ptr != null):
print ptr (this will print the path in reverse)
ptr = prev[ptr]

So this algorithm works because the topological sort always ensures that the graph is always tranversed from a source to a sink and because the sources of a DAG have the highest post order number they will always be some of the first vertices the algorithm vists and so the algorithm will work its way down the graph. The topological sort will take $O(|V| + |E|)$ time and so will the two for-loops in this algorithm which loop over all the vertices and all the edges. max and the while loop at the end are both $O(|V|)$. Therefore the algorithm is still $O(|V| + |E|)$.

5. Let's assume that all of the streets are reachable from all of the other streets and that streets form undirected edges between intersections. In other words, Sunnydale is strongly connected. Vertices can either be Unvisite or Visited and edges between two intersections v and u can be either Explored or Unexplored. Note that the state of edge (v, u) is the same as the state of edge (u, v). A vertex will be marked as Fully Visited once all of its neighboring edges have been explored. All vertices are initialized to Unvisited and all edges are initialized to Unexplored.
Here's pseudocode for my algorithm to before this search:
BuffyPatrol(v):
v.state = Visited
for each edge (v, u):
if (v, u).state == Unexplored:
(v, u).state = Explored
print (v, u)
(u, v).state = Explored
if u.state == Unvisited:
BuffyPatrol(u)
print (u, v)

So this algorithm runs in time $O(|V| + |E|)$ because every edge is traversed or at least checked to see if it has been traversed before and every vertex is explored at least once by the algorithm. This algorithm will work and will find Buffy a successful series of edges that she can walk accross so that she will be able to go up and down every street in the city exactly once. The order in which the edges are printed will give Buffy the list of edges she should take to complete her route in the way she wants. If we start at some vertex v we mark it as visited adn then explore all the edges that come out of it. If an unexplored edge leads to a vertex u that has already been visited, return to v and this will cover the edge connecting v and u and will always work. Now if u hasn't already been visited, we go to u and explore the edges coming out of u. The reason that edge (u, v) is marked 'Explored' before BuffyPatrol is called on u is so that when we are at u, we don't go back to v and leave u too soon. The order the edges are printed in is the order than buffy should go to complete her route. This process will run in $O(|V| + |E|)$ time.

6. Use an array, Q, of length $|V - 1| \cdot m + 1$ to keep track of the distances from the start vertex. Space i in the array will hold all the vertices that are i away from the start in a linked list. We only need

an array of this size because all the weights on the graphs are non-negative integers and the maximum edge weight is m and so the farthest a node could be from the start is $|V-1|m$ and because we need to keep track of distances between 0 and $|V-1|\cdot m$, the array needs to have $|V-1|\cdot m$ spaces. All i linked lists in Q are initialized to being empty, which makes sense because nothing has been pushed to them at the beginning.

So as in Dijkstra's algorithm, initialize another array, d, that will hold the distance each vertex is from the start so that every vertex's initial distance in infinite. Then add the start vertex onto $Q[0]$ and set d[Start] to 0. Now iterate over all spaces in the array Q in the following way, a process which will take $|V|m$ time. While there are still vertices in $Q[i]$, remove each vertex, v, from $Q[i]$, explore all of the edges that come out of v (when we do this over the whole graph this will add time $|E|$ to whole algorithm because we will hit every edge exactly once). if u is one of v's neighbors and $d[u] > d[v] + w(v,u)$ (i.e. going through v to get to u is a shorter path than the one we've already found), then remove u from $Q[d[u]]$, set $d[u]$ to $d[v] + w(v,u)$, and then add u to $Q[d[u]]$. We never need to worry about u getting put into a linked list in Q that has an index less than i (and so we might never be able to reach it) because the weights of the edges of the graph are all non-negative. Once all of v's neighbors have been explored and if v is the last element in it's linked list, increment i by 1 and continue to the next linked list. If we're at the end of the list, terminate the algorithm. This whole process takes $O(|E| + |V-1|m) = O(|E| + |V|m - m) = O(|E| + |V|m)$ time.

7. So while computing some of the values of $T(n)$ it noticed that the difference between $T(n+1)$ and $T(n)$ was always $\lceil \log_2(n+1) \rceil$. Therefore I concluded that $T(n) = \sum_{i=0}^{n} \lceil \log_2(n) \rceil$. Writing this sum out we see that
$= 0 + 1 + 2 + 2 + 3 + 3... + \lceil \log_2(n) \rceil \cdot (n - 2^{\lceil \log_2(n) \rceil - 1})$

$= 0 + 1 + 2 \cdot 2^1 + 3 \cdot 2^2 + 4 \cdot 2^3 + 5 \cdot 2^4 ... + \lceil \log_2(n) \rceil \cdot (n - 2^{\lceil \log_2(n) \rceil - 1})$

$= \lceil \log_2(n) \rceil \cdot (n - 2^{\lceil \log_2(n) \rceil - 1}) + \sum_{i=0}^{\lfloor \log_2(n) \rfloor} i \cdot 2^{i-1}$

Once we work through all the algebra, we end up with $T(n) = n\lceil \log_2 n \rceil + 2^{\lceil \log_2 n \rceil} + 1$
Now to prove that this is actually a valid explicit formula for the recurrence I'll use a proof by induction.
Base cases: $T(1) = 1\lceil \log_2 1 \rceil - 2^{\lceil \log_2 1 \rceil} + 1 = 0 - 1 + 1 = 0$.
$T(2) = 2\lceil \log_2 2 \rceil - 2^{\lceil \log_2 2 \rceil} + 1 = 2 - 2 + 1 = 1$.
Therefore the base cases hold.
Induction hypothesis: for all $m < n$, the equation $n\lceil \log_2 n \rceil + 2^{\lceil \log_2 n \rceil} + 1$ solves the recurrence equation given in the problem.
In order to prove this induction hypothesis, I will consider three cases: when n is even, when n is odd and one more than a power of 2, and when n is odd and not one more than a power of 2. Also, there are a few important rules about logs and the ceiling function that I will use throughout the proofs. $\lceil \log_2(b+1) \rceil = \lceil \log_2(b) \rceil$ if b is not a power of 2. Also, $k\lceil \log_2(2b) \rceil = k\lceil \log_2(b) \rceil + k$

Case: n is even. Let $n = 2b$:
$T(n) = T(b) + T(b) + n - 1$
$= 2b\lceil \log_2(b) \rceil - 2^{\lceil \log_2(b) \rceil + 1} + 2 + n - 1$
$= 2b\lceil \log_2(b) \rceil + 2b - 2^{\lceil \log_2(2b) \rceil} + 1$
$= 2b\lceil \log_2(2b) \rceil - 2^{\lceil \log_2(2b) + 1 \rceil}$
$= n\lceil \log_2(n) \rceil - 2^{\lceil \log_2(n) + 1 \rceil}$
and so the induction hypothesis holds if n is an even number.

Case: $n = 2^k + 1$

$T(n) = T(2^k + 1) = T(\lceil 2^k + 1 \rceil / 2) + T(\lfloor 2^k + 1 \rfloor / 2) + 2^k + 1 - 1$
$= T(2^{k-1} + 1) + T(2^{k-1}) + 2^k$
$= (2^{k-1} + 1)k - 2^k + 1 + (2^{k-1})(k-1) - 2^{k-1} + 1$
$= k2^k - 2^k + k + 2$
Now to check that the induction hypothesis still holds, I will plug $2^k + 1$ directly into the induction hypothesis.
$T(2^k + 1) = (2^k + 1)(k+1) - 2^{k+1} + 1$
$= k2^k + 2^k + k + 1 - 2^{k+1} + 1 = k2^k - 2^k + k + 2$ which confirms the prior result.

Case: $n = 2b + 1$ such that $b \neq 2^k$
$T(n) = T(b+1) + T(b) + n - 1$
$= (b+1)\lceil \log_2(b+1) \rceil - 2^{\lceil \log_2(b+1) \rceil} + 1 + b\lceil \log_2(b) \rceil - 2^{\lceil \log_2(b) \rceil} + 1 + 2b$
$= 2b\lceil \log_2(b) \rceil + 2b + \lceil \log_2(b) \rceil + 1 - 2^{\lceil \log_2(2b) \rceil} + 1$
$= (2b+1)\lceil \log_2(2b) \rceil - 2^{\lceil \log_2(2b) \rceil} + 1$
$= n\lceil \log_2(n) \rceil - 2^{\lceil \log_2(n) \rceil} + 1$ and so the induction hypothesis is true and the result is proved.
$T(n) = n\lceil \log_2(n) \rceil - 2^{\lceil \log_2(n) \rceil} + 1$

I collaborated with Manav Khandelwal and Lauren Kim on this pset.