CS 124 Problem Set 2
HARVARD ID: 10939860

1. Buffy and Willow are facing an evil demon named Stooge, living inside Willow?s computer. In an effort to slow the Scooby Gang?s computing power to a crawl, the demon has replaced Willow?s hand-designed superfast sorting routine with the following recursive sorting algorithm, known as StoogeSort. For simplicity, we think of Stoogesort as running on a list of distinct numbers. StoogeSort runs in three phases. In the first phase, the first 2/3 of the list is (recursively) sorted. In the second phase, the final 2/3 of the list is (recursively) sorted. Finally, in the third phase, the first 2/3 of the list is (recursively) sorted again.

   Willow notices some sluggishness in her system, but doesn?t notice any errors from the sorting routine. This is because StoogeSort correctly sorts. For the first part of your problem, prove rigorously that StoogeSort correctly sorts. (Note: in your proof you should also explain clearly and carefully what the algorithm should do and why it works even if the number of items to be sorted is not divisible by 3. You may assume all numbers to be sorted are distinct.) But StoogeSort can be slow. Derive a recurrence describing its running time, and use the recurrence to bound the asymptotic running time of Stoogesort.

   Solution.
   We can prove the correctness of StoogeSort by induction. We can easily see that the given algorithm works for lists of length 1 and 2, so we will let a list of length 2, $[i, j]$, be our base case. We divide the range of list $[i, j]$ into 3 equal parts, a, b, and c.

   After the first recursion, sorting the first 2/3 of the list, or $[xy]$, every element in segment y is larger than every element in segment x after sorting. Next, after the second recursion, sorting the final 2/3 of the list, or $[yz]$, all of the list's largest elements have moved to segment z, meaning z has the list's largest elements and is sorted. Next, after the final recursion of the first 2/3 of the list again, $[xy]$ is sorted, making the entire list sorted. We can thus conclude that the entire list is sorted, because each of the 3 subsections are sorted and the elements in z are larger than x and y, while the elements in y are larger than x. For this algorithm to work, the thirds must be rounded up on lists with length not a multiple of 3 (ex: 2/3 of 8 is 4, not 3). If the thirds are rounded down, segment z will have the largest number of elements, and the algorithm will fail.

   The algorithm calls a constant-time operation, and then recursively calls itself 3 times. Each instance of recursion involves calling the function on a list 2/3 the size of the given list of length n. Thus, a recurrence relation for StoogeSort is as follows: $T(n) = 3T(\frac{2}{3}n) + 1$. We can then use the Master Theorem for recurrences to solve this relation. In the stated recurrence, $a = 3, b = \frac{3}{2}, c = 1$, and $k = 1$. Thus, since $a > b^k$, the Master Theorem says $T(n) = O(n^{log_b(a)}) = O(n^{log_{3/2}(3)}) = \Theta(n^{2.71})$.

2. (Part A) Solve the following recurrences exactly, and then prove your solutions are correct by induction. (Hint: Graph values and guess the form of a solution: then prove that your guess is correct.)

   - $T(1) = 1, T(n) = T(n-1) + 4n - 4$
   - $T(1) = 1, T(n) = 2T(n-1) + 2n - 1$

   (Part B) Give asymptotic bounds for T(n) in each of the following recurrences. Hint: You may have to change variables somehow in the last one.

- $T(n) = 4T(n/2) + n^3$
- $T(n) = 17T(n/4) + n^2$
- $T(n) = 9T(n/3) + n^2$
- $T(n) = T(\sqrt{n}) + 1$

Solution.

(Part A)

- Because of the $T(n-1)$ term, guess that the solution to the recurrence is a quadratic: $T(n) = an^2+bn+c$. We substitute in the given recurrence to get $T(n) = a(n-1)^2+b(n-1)+c+4n-4$, which calculates to a final solution of $T(n) = 2n^2 - 2n + 1$ (using the fact that $a+b+c = 1$). To prove this, we use induction. Clearly, the base case $T(1) = 1$ is true: $T(1) = 2 - 2 + 1 = 1$. Next, assuming $T(n-1)$ to be true, we prove the case of $T(n-1+1) = T(n)$. Using the given recurrence, $T(n) = T(n-1) + 4n - 4 = 2(n-1)^2 - 2(n-1) + 1 + 4n - 4 = 2n^2 - 2n + 1$. The inductive answer is the same as the original calculated answer, so this solution is true for all n.
- Here, due to the 2 coefficient before the $T(n-1)$ term, we guess an exponential solution added to a quadratic (for the same reasons as the previous question): $T(n) = a2^n + bn^2 + cn + d$. Doing algebra and reaching a solution using the fact that $2a + b + c + d = 1$, we get the result $T(n) = 3(2^n) - 2n - 3$ as our estimated solution. For induction, we see that the base case is true: $T(1) = 3(2) - 2(1) - 3 = 1$. To prove this works for all n, we plug in the given recurrence, assuming $T(n-1)$ works: $T(n) = 2T(n-1) + 2n - 1 = 2(3(2^{n-1}) - 2(n-1) - 3) + 2n - 1 = 6(2^{n-1}) - 2n - 3 = 3(2^n) - 2n - 3$. The answers are the same, meaning are induction has proved true.

(Part B)

- By the master theorem, $a = 4, b = 2, k = 3$, and since $a < b^k$, this recurrence has asymptotic bound $\Theta(n^3)$.
- By the master theorem, $a = 17, b = 4, k = 2$, and since $a > b^k$, this recurrence has asymptotic bound $\Theta(n^{log_4 17})$.
- By the master theorem, $a = 9, b = 3, k = 2$, and since $a = b^k$, this recurrence has asymptotic bound $\Theta(n^2 logn)$.
- Let $n = 2^m$. Then $T(n) = T(2^m) = T(2^{m/2})+1$. Next, let $T(2^m) = S(m)$, where S is some other function of m. Then $S(m) = S(m/2)+1$. By the master theorem, since $a = b^k$, $S(m) = log(m)$. Because we let $n = 2^m$, we have $m = log(n)$, meaning $T(n) = S(log(n)) = log(log(n))$.

3. Explain how to solve the following two problems using heaps. (No credit if you're not using heaps!) First, give an $O(n \log k)$ algorithm to merge k sorted lists with n total elements into one sorted list. Second, say that a list of numbers is k-close to sorted if each number in the list is less than k positions from its actual place in the sorted order. (Hence, a list that is 1-close to sorted is actually sorted.) Give an $O(n \log k)$ algorithm for sorting a list of n numbers that is k-close to sorted.

Solution.

For the first part, consider the following algorithm: first, put the first element of each list into a min-heap using $BUILD - HEAP$, while keeping track of the list L_i that each element belongs to. Next, run $EXTRACT{-}MIN$ on the min-heap and store it in a separate *sorted* array, and then switch

out the next element from the list L_i into the heap. Repeat this process, running $MIN-HEAPIFY$ in each iteration to maintain a min-heap, until the heap is empty and $EXTRACT-MIN$ will not run anymore, meaning there are no more elements in any of the initial lists, and the *sorted* array is completely sorted.

Consider the following example to illustrate this algorithm:
Input: $[1, 8], [2, 4], [5, 6]$
Initial heap: $[1, 2, 5]$

Step 1: extract 1, insert 8
sorted : $[1]$
Heap: $[2, 5, 8]$

Step 2: extract 2, insert 4
sorted : $[1, 2]$
Heap: $[4, 5, 8]$

Step 3: extract 4, insert nothing
sorted : $[1, 2, 4]$
Heap: $[5, 8]$

Step 4: extract 5, insert 6
sorted : $[1, 2, 4, 5]$
Heap: $[6, 8]$

Step 5: extract 6, insert nothing
sorted : $[1, 2, 4, 5, 6]$
Heap: $[8]$

Step 6: extract 8, insert nothing
sorted : $[1, 2, 4, 5, 6, 8]$
Heap: $[]$

Complete.

First, the algorithm runs $BUILD-HEAP$, which will take $O(klog(k))$ time. At each step, the algorithm will call $MIN-HEAPIFY$ once, and each iteration will take $O(log(k))$ time, since we are taking one element from each k for the heap. The iteration will run n times, since in each step a node is popped off and a new node is added to the heap, running $MIN-HEAPIFY$ again, and there are n total elements in the lists, coming out to $nlog(k))$ time. Similarly, on each step $EXTRACT-MIN$ will run and take $O(nlog(k))$ time in total. The total running time comes to $O(klog(k) + 2nlog(k)) = O(nlog(k))$.

For the second part, we use an adapted algorithm that ultimately has the same running time. First, run $BUILD-HEAP$ on the first k elements in the list. Since the numbers are k-close to sorted, we can guarantee that the absolute minimum of the list is within the first k elements, and the pattern continues. After building the initial min-heap, run $EXTRACT-MIN$ to get the minimum number and put it in a separate *sorted* list. Then, replace the deleted node in the heap with the next number

in the list, and run $MIN - HEAPIFY$ to resort into a min-heap, and repeat the algorithm until the heap is empty, then return the *sorted* list.

Running time: First, we build the initial heap, which takes $O(klog(k))$ time. Then, we run $MIN - HEAPIFY$ n times, each iteration with a running time of $log(k)$, coming out to $O(nlog(k))$. Then, we extract the minimum number in the heap, which takes $O(log(k))$ time, and we do that n times, coming out to $O(nlog(k))$. So, the total running time is $O(2nlog(k) + klog(k)) = O(nlog(k))$.

4. Design an efficient algorithm to find the longest path in a directed acyclic graph. (Partial credit will be given for a solution where each edge has weight 1; full credit for solutions that handle general real-valued weights on the edges, including negative values.)

   Solution.

   Since we are given a directed acyclic graph, we can use topological sort to to give us a list of the given graph's nodes in a sorted order that will tell the algorithm what order to traverse the nodes. Topological order will give a valid directional path of how to accurately determine the weights of each edge. Initialize a *lengths* array of size $|V|$ where $lengths[i]$ will give the length from the start node to node i. Iterate through the set of vertices in topological order for each vertex v, and then iterate through each edge (v, w). Within those loops, check if $lengths[w] <= lengths(v) + weight(v, w)$. If this statement is true, then update: $lengths[w] = lengths(v) + weight(v, w)$. This recursion ensures that each time, the step that will yield the longest path is recorded, and every vertex will be checked, recursing every node on each level before moving to the next level. When the if statement is satisfied, we update the total length from the start node to that particular vertex. Then, at the end of the function, we return the maximum value $lengths[i]$ in the *lengths* array. This will give us the length of the longest path.

   Now we consider the runtime. First, we include the time of topological sort, known to be $O(|V|+|E|)$. Then, in the algorithm's loops, it traverses and checks each vertex and each edge once, or $O(|V|+|E|)$. Finally, we return the max value in the *lengths* array, a function that will take $O(|V|)$ time in the worst case. Thus, the total running time is $O(|V| + |E| + |V| + |E| + |V|) = O(|E| + |V|)$.

5. Giles has asked Buffy to optimize her procedure for nighttime patrol of Sunnydale. (This takes place sometime in Season 2.) Giles points out that proper slaying technique would allow Buffy to traverse all of the streets of Sunnydale in such a way that she would walk on each side of each street, exactly once, going up the street in one direction and down the street in the other direction. Buffy now has slayer homework: how can it be done? (If you have to assume anything about the layout of the city of Sunnydale, make it clear!)

   Solution.

   Buffy's mission can be solved simply with a modified version of depth-first search. DFS ensures that every edge in a graph is visited twice, so Buffy must keep track of which streets have been already traversed. Buffy assumes that the graph is all strongly connected. With DFS, Buffy will traverse each new, unvisited node, and mark it as visited once (marking only one side of the street?the right side, abiding by general pedestrian rules). Then, once she has visited each node once, she will detect that there are no more nodes to visit that she hasn't been to. Then, DFS tells her to work her way backwards, continuing to traverse each node once again, in a similar way, marking the other

side of each street as visited until she reaches the start node again. This mean that every edge (street) is visited exactly twice. Just like typical DFS, the running time of Buffy's algorithm will be $O(|V| + |E|)$.

6. Consider the shortest paths problem in the special case where all edge costs are non-negative integers. Describe a modification of Dijkstra's algorithm that works in time $O(|E| + |V|m)$, where m is the maximum cost of any edge in the graph.

   Solution.

   We know that the typical runtime of Dijkstra's algorithm is $O(|V|*DELETEMIN+|E|*INSERT)$, so we will need to create a modification such that $DELETEMIN$ takes $O(m)$ time and $INSERT$ takes $O(1)$ time.

   Let us consider an alternate implementation of the priority queue or heap. We will use a hash table. Create an array A of keys that are entries that are subsets of V. The array will be of size $|V|m$, since this it the longest possible path through the graph. We choose this length because values in the *dist* array are the lengths of paths of at most $|V| - 1$ edges, meaning no path is longer than $|V|m$. Each key in the array will point to a value that is a linked list containing the vertices that have the distance of that key (for example: if i and j are nodes each with length 6, the key 6 in A will point to a linked list containing nodes i and j. Each time we insert a new value $dist[v] + length(v, w)$ into the algorithm's priority queue as in Dijkstra's algorithm, it is greater than the minimum value $dist[v]$ deleted at the beginning of the function call. When we need to add a new pair of the node and its corresponding distance into the data structure, we can simply find the key of the desired distance, and add the node into the linked list that it points to. Since inserting into a linked list takes constant time, this modified version of $INSERT$ will run in $O(1)$ time. $DELETEMIN$ can be achieved by searching across the array of keys for the first nonempty entry $A[i]$, and returning the first element of the key's corresponding value (linked list), or any element in it. Since linked lists have $O(n)$ 5access time, and distances returned can be anywhere from 0 to m, the runtime for $DELETEMIN$ will be $O(m)$ in the worst case. Thus, the total runtime is $O(|E| + |V|m)$.

7. (Note: this problem is somewhat difficult) We found that a recurrence describing the number of comparison operations for a mergesort is $T(n) = 2T(n/2) + n - 1$ in the case where n is a power of 2. (Here $T(1) = 0$ and $T(2) = 1$.) We can generalize to when n is not a power of 2 with the recurrence

$$T(n) = T(\lceil n/2 \rceil) + T(\lfloor n/2 \rfloor) + n - 1$$

   Exactly solve for this more general form of $T(n)$, and prove your solution is true by induction. (Additional note: you will not receive full credit if your answer is in the form of a summation over say n terms. We are looking for a closed form answer.) Hint: plot the first several values of $T(n)$, graphically. What do you find? You might find the following concept useful in your solution: what is $2^{\lceil \log 2n \rceil}$?

   Solution.

   To solve this recurrence, we first plot around 20 of the points on a graph to get a sense of the solution pattern, and guess that the solution is $T(n) = an\lceil log_2(n) \rceil - b2^{\lceil log_2(n) \rceil} + c$. Using algebra, we find coefficients a, b, and c are all equal to 1, so our closed form solution is $T(n) = n\lceil log_2(n) \rceil - 2^{\lceil log_2(n) \rceil} + 1$.

Now, we will prove the solution true by induction. The base cases are evidently true: $T(1) = 1 * 0 - 2^0 + 1 = 0$, $T(2) = 2 * 1 - 2^1 + 1 = 1$. For the inductive step, we assume that the solution works for n from 1 to $n-1$, and we will now prove for n. We split the induction into three cases where the process for proving each have different simplifications for the floor and ceiling functions: Case 1 is for even n, Case 2 is for odd n where $n-1$ is a power of 2, and Case 3 is for odd n where $n-1$ is not a power of 2. See the following inductive algebra for each case.

Case 1:
$T(n) = T(\lceil n/2 \rceil) + T(\lfloor n/2 \rfloor) + n - 1 = 2T(n/2) + n - 1$
$= 2(n/2\lceil log_2(n/2) \rceil - 2^{\lceil log_2(n/2) \rceil} + 1) + n - 1$ (because n is even)
$= n\lceil log_2(n/2) \rceil - 2^{\lceil log_2(n/2) \rceil + 1} + n + 1$
$= n\lceil log_2(n) - log_2(2) \rceil - 2^{\lceil log_2(n) - log_2(2) + log_2(2) \rceil} + n + 1$
$= n\lceil log_2(n) - 1 \rceil - 2^{\lceil log_2(n) \rceil} + n + 1$
$= n\lceil log_2(n) \rceil - n - 2^{\lceil log_2(n) \rceil} + n + 1$
$= n\lceil log_2(n) \rceil - 2^{\lceil log_2(n) \rceil} + 1$

Case 2:
let $n = 2k + 1$, where k is an integer
$T(n) = T(\lceil (2k+1)/2 \rceil) + T(\lfloor (2k+1)/2 \rfloor) + n - 1$
$= T(\lceil k + 1/2 \rceil) + T(\lfloor k + 1/2 \rfloor) + n - 1$
$= T(k) + T(k+1) + n - 1$
$= k\lceil log_2(k) \rceil - 2^{\lceil log_2(k) \rceil} + 1 + (k+1)\lceil log_2(k+1) \rceil - 2^{\lceil log_2(k+1) \rceil} + 1 + n - 1$
$= k\lceil m \rceil - 2^{\lceil m \rceil} + (k+1)\lceil m + 1 \rceil - 2^{\lceil m+1 \rceil} + n + 1$ (where $m = log_2(k)$)
$= k\lceil m \rceil + k\lceil m + 1 \rceil + \lceil m + 1 \rceil - 2^{\lceil m \rceil} - 2^{\lceil m+1 \rceil} + n + 1$
$= 2k\lceil m \rceil + m + 1 - 3(2^{\lceil m \rceil}) + n + 1$
$= m(2k+1) + k + 1 - 3(2^m) + 2k + 2$
$= m(2^m + 1) + 2^m - 3(2^m) + 2(2^m) + 3$
$= m2^m + m + 2^m + 3$
$= (2^{m+1})m + m + 3$
$= log_2(k)(2k+1)$, which simplifies to:
$= n\lceil log_2(n) \rceil - 2^{\lceil log_2(n) \rceil} + 1$

Case 3:
$T(n) = T(\lceil n/2 \rceil) + T(\lfloor n/2 \rfloor) + n - 1$
$= T(\lceil k + 1/2 \rceil) + T(\lfloor k + 1/2 \rfloor) + n - 1$
$= T(k) + T(k+1) + n - 1$
$= k\lceil log_2(k) \rceil - 2^{\lceil log_2(k) \rceil} + 1 + (k+1)\lceil log_2(k+1) \rceil - 2^{\lceil log_2(k+1) \rceil} + 1 + 2k + 1 - 1$
$= k\lceil log_2(k) \rceil - 2^{\lceil log_2(k+1) \rceil} + k\lceil log_2(k) \rceil + \lceil log_2(k) \rceil + 2k + 2$
$= (2k+1)\lceil log_2(k) \rceil - 2^{\lceil log_2(k+1) \rceil} + 2k + 1 + 1$
$= (2k+1)(\lceil log_2(k) \rceil - 2^{\lceil log_2(k) + log_2(2) \rceil} + 2k + 1 + 1 = (2k+1)(\lceil log_2(k) \rceil - 2^{\lceil log_2(2k) \rceil} + 2k + 1 + 1$,
which simplifies to:
$= n\lceil log_2(n) \rceil - 2^{\lceil log_2(n) \rceil} + 1$

Therefore, the induction is satisfied for all cases, and $T(n) = n\lceil log_2(n) \rceil - 2^{\lceil log_2(n) \rceil} + 1$ is the correct solution to this recurrence.

I collaborated on this problem set with Kate Schember, Jeremy Welborn, and Nathan Lee.