Problem Set 2

Harvard ID: 30940040

For all homework problems where you are asked to give an algorithm, you must prove the correctness of your algorithm and establish the best upper bound that you can give for the running time. You should always write a clear informal description of your algorithm in English. You may also write pseudocode if you feel your informal explanation requires more precision and detail. As always, try to make your answers as clear and concise as possible.

1. **Buffy and Willow are facing an evil demon named Stooge, living inside Willows computer. In an effort to slow the Scooby Gangs computing power to a crawl, the demon has replaced Willows hand-designed super- fast 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 doesnt 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.

   Stooge Sort Algorithm
   Given. No need to modify.

   Proof of Correctness
   We will prove the following through induction. We first need to prove the base case, which is that the algorithm will correctly sort for 1, 2, and 3 element cases. These cases are trivial. For the 1 element case, it is already sorted. For the 2 element case, the first pass of 2/3 of the elements properly sorts. For the 3 element case, the first pass sorts the first two elements, which puts the largest element between the two first elements in the middle. The second pass puts the largest element in the 3rd spot (properly sorted). and the final pass puts the smallest element in the 1st spot, and thus the element in the middle is properly sorted.

   Now we will prove that Stooge Sort sorts correctly by using strong induction on the size of the list. Assume that Stooge Sort will sort lists of size $1 \le i \le n$ correctly, where any list of size i will be sorted correctly. We now prove that Stooge Sort will sort a list of size $n+1$ correctly. We begin by sorting the first 2/3 of the list. Let's call these thirds X, Y, and Z respectively, and plugging in $n+1$ we say that $|X| = |Y| = \lceil \frac{n+1}{3} \rceil$. The ceiling is used because it will allow us to split a list into thirds a list who's size is not divisible by 3. After we sort the first $\frac{2}{3}$, we know that every element in $X < Y$, by the inductive hypothesis. We also know that $|XY| = 2\lceil \frac{n+1}{3} \rceil$. Comparing this to the size of our total list we have:

$$\lceil \frac{2n+2}{3} \rceil \le n+1$$

$$\lceil \frac{2n}{3} + \frac{2}{3} \rceil \le n+1$$

$$\lceil\frac{2n}{3}\rceil + \lceil\frac{2}{3}\rceil \leq n+1$$

We know by the properties of ceilings:

$$\lceil\frac{2n}{3}\rceil + \lceil\frac{2}{3}\rceil \geq \lceil\frac{2n}{3} + \frac{2}{3}\rceil$$

So we can safely substitute:

$$\lceil\frac{2n}{3}\rceil + 1 \leq n+1$$

And the following is always true:

$$\lceil\frac{2n}{3}\rceil \leq n$$

Therefore, using ceilings is a valid way to deal with list sizes that are not multiples of 3. Now we continue the proof, we sort the next $\frac{2}{3}$ of the list, and know by by the inductive hypothesis that all elements in $Y < Z$. And by transitivity, since all $X < Y$, we know that we know have the $\lceil\frac{n+1}{3}\rceil$ largest elements, in sorted order, in Z. Finally, we sort the X and Y again such that all elements in $X < Y$ by the I.H., and thus by transitivity since $Y < Z$ we have the $2\lceil\frac{n+1}{3}\rceil$ smallest elements in sorted order in XY. Therefore, the entire list has been sorted, and we have proved this by induction. QED.

Solving the Recurrence
We know the algorithm will recursively call itself on $\frac{2}{3}$ of the list three times, performing the constant time operations of comparing and swapping two elements, which we say is takes time of $O(1) = 1$. Thus we have $T(n) = 3T(\frac{2}{3}n) + 1$. The worst case runtimes for the base cases are then $T(1) = 0, T(2) = 1$, and $T(3) = 3$. Using the recurrence tree method from CLRS, we unwind the tree and see a pattern of $3^0, 3^1, 3^2 ... 3^k$ operations. We don't need to deal with the exact terms in this method, as CLRS indicates, as it is good for finding an upper bound. We can then conclude that the tree is growing at a cubic rate at the fastest, and thus we have $O(n^3)$.

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.)**

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

     Solution:

     Solution to the Recurrence
     The table below shows the first 5 values of the recurrence. Plotting it indicated a quadratic looking curve, so we guess that our solution will be of the form T$(n) = an^2 + bn + c$.

n	T(n)
1	1
2	5
3	13
4	25
5	41

Plugging in our guess:

T$(n) = a(n-1)^2 + b(n-1) + c + 4n - 4$
$= an^2 - 2an + a + bn - b + c + 4n - 4$
$= an^2 + n(-2a + b + 4) + (a - b + c - 4)$

Matching up the coefficients:

$a = a$
$b = -2a + b + 4$
$c = a - b + c - 4$

We solve the 2^{nd} equation to get $a = 2$, and the 3^{rd} to get $b = -2$. By plugging in a and b into the initial conditions, T$(1) = a - b + c - 4 = 1$, we get $c = 1$. Plugging it into our guess form we get the following closed form solution:

T$(n) = 2n^2 - 2n + 1$
$= 2n(n-1) + 1$

Proof by Induction
We will prove by induction that T(n) = $2n(n-1)+1$ is the solution to the recurrence T(n) = T$(n-1)$ + $4n - 4$ for any $n \geq 1$. We begin by confirming that the base case, $T(1) = 1$, is satisfied by the solution. T$(1) = 2(1)(1-1) + 1 = 1$, so this works.

Through the inductive hypothesis, we can assume that T$(n) = 2n(n-1)+1$ is true for $n = k$. We will prove that it is true for $n = k+1$. Plugging in $k+1$ into our solution gives us:

T$(k+1) = 2(k+1)(k+1-1) + 1$
$= 2k(k+1) + 1$

Now we plug $k+1$ into our recurrence, and simplify. Since we assume T$(n) = 2n^2 - 2n + 1$ to be true, we can substitute it into our equation:

T$(k+1)$ = T$(k) + 4(k+1) - 4$
= T$(k) + 4k$
$= 2k^2 - 2k + 1 + 4k$
$= 2k(k+1) + 1$

We see that the original recurrence equals the solution for $n = k+1$, and thus we have proved our solution is correct through induction. QED.

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

  Solution:

  Solution to the Recurrence

The table below shows the first 5 values of the recurrence. We see that the recurrence has a $2\mathrm{T}(n-1)$ in it, which indicates that there will likely be an exponential term of 2^n in the solution. It also has an n term and constant term that will be continuously accumulating through addition, so we guess the form will be $\mathrm{T}(n) = a2^n + bn + c$.

n	$\mathrm{T}(n)$
1	1
2	5
3	15
4	37
5	83

We arrive at the solution $\mathrm{T}(n) = 3 \times 2^n - 2n - 3$ with the help of Wolfram Alpha. It was indeed the form we guessed. (Piazza post @175 indicates we need only show the induction proof for the solution).

Proof by Induction
We will prove by induction that $\mathrm{T}(n) = 3 \times 2^n - 2n - 3$ is the solution to the recurrence $\mathrm{T}(n) = 2T(n-1) + 2n - 1$ for any $n \geq 1$. We begin by confirming that the base case, $T(1) = 1$, is satisfied by the solution. $\mathrm{T}(1) = 3 \times 2^1 - 2(1) - 3 = 1$, so this works.

Through the inductive hypothesis, we can assume that $\mathrm{T}(n) = 3 \times 2^n - 2n - 3$ is true for $n = k$. We will prove that it is true for $n = k + 1$. Plugging in $k + 1$ into our solution gives us:

$\mathrm{T}(k+1) = 3 \times 2^{k+1} - 2(k+1) - 3$
$= 3 \times 2^{k+1} - 2k - 5$

Now we plug $k + 1$ into our recurrence, and simplify. Since we assume $\mathrm{T}(n) = 3 \times 2^n - 2n - 3$ to be true, we can substitute it into our equation:

$\mathrm{T}(k+1) = 2T(k) + 2(2k+1) - 1$
$= 2T(k) + 2k + 1$
$= 2(3 \times 2^k - 2k - 3) + 2k + 1$
$= 3 \times 2^{k+1} - 4k - 6 + 2k - 1$
$= 3 \times 2^{k+1} - 2k - 5$

We see that the original recurrence equals the solution for $n = k + 1$, and thus we have proved our solution is correct through induction. QED.

(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.

- $T(n) = 4T(n/2) + n^3$

  Solution:

We will use the Master Theorem to get asymptotic bounds. We let $a = 4$, $b = 2$, and $k = 3$. Since $4 < 2^3$, we use the third case of the theorem and get $\mathrm{T}(n) = \Theta(n^3)$.

- $T(n) = 17T(n/4) + n^2$

  Solution:

  We will use the Master Theorem to get asymptotic bounds. We let $a = 17$, $b = 4$, and $k = 2$. Since $17 > 4^2$, we use the first case of the theorem and get $\mathrm{T}(n) = \Theta(n^{\log_4 17})$.

- $T(n) = 9T(n/3) + n^2$

  Solution:

  We will use the Master Theorem to get asymptotic bounds. We let $a = 9$, $b = 3$, and $k = 2$. Since $9 = 3^2$, we use the second case of the theorem and get $\mathrm{T}(n) = \Theta(n^2 \log n)$.

- $T(n) = T(\sqrt{n}) + 1$

  Solution:

  We will use the Master Theorem to get asymptotic bounds. We begin with:

  $\mathrm{T}(n) = \mathrm{T}(\sqrt{n}) + 1$
  $= T(n^{\frac{1}{2}}) + 1$

  We need to find an appropriate way to bring this $\frac{1}{2}$ down to make it usable for the Master Theorem, so we need to find a substitution that will be easy to manipulate. Logarithms and exponents work well, so we let $n = e^k$. This gives us:

  $\mathrm{T}(n) = \mathrm{T}(n^{\frac{1}{2}}) + 1$
  $= \mathrm{T}(e^k) = T(e^{\frac{k}{2}}) + 1$

  The logarithm substitution allows us to easily take advantage of the inverse $k = \ln n$. We create a new function $X(k)$ to use this substitution, which will allow us to calculate Master's Theorem, and then convert back to $T(n)$:

  $\mathrm{X}(k) = \mathrm{T}(\ln e^k)$
  $= \mathrm{T}(\ln e^{\frac{k}{2}}) + \ln 1 = \mathrm{X}(\frac{k}{2}) + 1$

  We can now use the Master Theorem to get asymptotic bounds for $\mathrm{X}(\frac{k}{2}) + 1$. We let $a = 1$, $b = 2$, and $k = 0$. Since $1 = 2^0$, we use the second case of the theorem and get $\mathrm{X}(k) = \Theta(\log k)$. Since we know $k = \ln n$, we now substitue for k in our runtime equation to get $\mathrm{T}(n) = \Theta(\log \log n)$.

3. **Explain how to solve the following two problems using heaps. (No credit if youre 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:

Note: We will operate under the assumption that our k sorted lists are each linked lists to have constant time 'popping' of the first element. If we are sorting in ascending order, our lists will be sorted in ascending order. If we are sorting in descending order, our lists will be sorted in descending order.

Algorithm to Merge k Sorted Lists with n Total Elements into One Sorted List
Step 1) Create an array A of size n which will store the results, and an array H of size k to be used as a heap. To sort in ascending order, this will be a min heap, to sort in descending order, use a max heap. Both heaps operate identically, except that a max heap has large-to-small from top-to-bottom, and a min heap has small-to-large from top-to-bottom. **To reduce redundancy, we will describe the algorithm in terms of a min heap, so ascending order. To sort in descending order, replace every instance of a "minimum" below with "maximum"!** We refer to the k sorted lists as $L_1, L_2, ...L_k$, and they must all be non-empty. Otherwise, remove empty lists and adjust k to reflect the number of non-empty lists. We also initialize a counter, $j = 0$, and a variable $prev = 0$.

Step 2) Pop the first element of each L_i, and insert them into H as tuples (e, i), where e is the integer element, and i is a number $1 \leq i \leq k$ denoting the list it's from.

Step 3) Extract the minimum element from H, and insert its e at $A[j]$, store it's i in $prev$, and increment j by 1.

(a) Pop the next element from L_{prev} and insert it into H. If L_{prev} and H are empty, end. If L_{prev} is empty, but H is not, start from the beginning of Step 3 again.

Proof of Correctness
Step 1): We have n total elements, so we know that an array of size n is sufficient to store all results. We also have k sorted lists - our intention is to store no more than k elements in H at any time, and we will see that invariant will hold in Steps 2 and 3. Step 1 is correct.

Step 2): We know that when we pluck the first element of each L_i, the smallest element $\forall n$, e_{min}, must be contained in this set of k first elements. We prove this by contradiction: Assume that e_{min} is not contained in this set of first elements from each L_i. This means that e_{min} is not at the head of any L_i. However since the lists are all sorted, each contains their smallest e at their head. And we know that e_{min} is smaller than or equal to all other e, so this is a contradiction and it must have been the head of some L_i.

We also know that we have k sorted lists, so in our initial insertion the binary heap will have k elements, because we only select the first element from each list. At this phase we do not have to deal with empty lists, as we eliminated them in Step 1 of our algorithm, so the number of elements will equal k. Step 2 is correct.

Step 3): Let q denote the number of non-empty k sorted lists, and remember e_{min} denotes the smallest element $\forall$ remaining n. In this phase we have to deal with 3 primary cases. The case where we have $q = k$, the case where we have $1 \leq q < k$, and the case where we have $q = 0$. In all cases, we will always have e_{min} in H after popping the head of an L_i, until H is empty.

In the case where we have $q = k$, we have shown that the initial insertion in Step 2 will have e_{min} in H. After we extract e_{min} from H, H will have $k - 1$ elements, which contain the smallest element from each L_i, except for L_{prev}. Then we proceed to insert the smallest element from L_{prev}, and we again have the smallest element from every list, and thus have e_{min} in H. For this case, the invariant will hold, as we will either have $k - 1$ elements after extracting the minimum from H, or k elements after inserting the minimum from L_{prev}.

In the the case where we have $1 \leq q < k$, the invariant will also hold. We will always have q or $q - 1$ elements in the heap. First we extract the minimum element from the heap, and we have $q - 1$ elements in H. H, therefore, contain the smallest element from each of the q remaining L_i, except for L_{prev}. We then check L_{prev} for the next element. If L_{prev}, is not empty, we take the smallest value from it and insert it into the list, therefore still retaining the invariant that we will have e_{min} in H when we have q elements in H. If L_{prev} is empty, then we have exhausted all elements in that list, so there will be $q - 1$ remaining lists (so by definition, q implicitly becomes $q - 1$). H still contains e_{min}, because it has the smallest element of each remaining L_i. H then extracts the minimum element, and repeats for the next L_{prev}. Every time a list is exhausted, $|H|$ decreases by one permanently, thus still containing the minimum element from each remaining, which implies it contains e_{min}.

For the final case where we have $q = 0$, then all all lists have been exhausted. There must therefore either be 1 element, which is $e_m in$ by definition, or no elements, in which case $|H| = q = 0$, and the algorithm halts. We have proven that the algorithm is correct. QED.

Analysis of Runtime
Step 1: First we create A of size n, which takes $O(n)$ time. Then we create H, which takes $O(k)$ time. This step is $O(k + n)$, which simplifies to $O(n)$.

Step 2 and 3: We will need to pop all n elements from the k sorted lists, which takes $O(n)$ time and insert them into H, which takes $O(n \log k)$ time. We will also extract all n elements from H, every time extracting the minimum, which takes $O(n \log k)$ time. Finally, we also need to insert all elements into the array A, which takes $O(n)$ time. Combining this with the first step we have $O(n + n + n \log k + n \log k + n)$ which simplifies to $O(n \log k)$ time.

Solution:

Note: We will use operate under the assumption that our k-close to sorted list is a linked list to have constant time 'popping' of the first element. If we are sorting in ascending order, our list will be sorted in ascending order. If we are sorting in descending order, our list will be sorted in descending *order.*

Algorithm to Sort a List of n Numbers that is k-Close to Sorted
Step 1) We call the k-close to sorted list T. Create an array A of size $n = |T|$ which will store the results, and an array H of size k to be used as a heap. To sort in ascending order, this will be a min heap, to sort in descending order, use a max heap. Both heaps operate identically, except that a max heap has large-to-small from top-to-bottom, and a min heap has small-to-large from top-to-bottom. **To reduce redundancy, we will describe the algorithm in terms of a min heap, so ascending order. To sort in descending order, replace every instance of a ”minimum” below with ”maximum”!** We also initialize a counter, $j = 0$.

Step 2) Pop the first k elements from T, and insert them into H.

Step 3) Extract the minimum element from H, insert it at $A[j]$, and increment j by 1.

(a) Pop the next element from T and insert it into H. If T and H are empty, end. If T is empty, but H is not, start from the beginning of Step 3 again, and do not repeat (a).

Proof of Correctness
Step 1): We have $|T| = n$ total elements, so we know that an array of size n is sufficient to store all results. We also have k elements in H at any time, and we will see that invariant will hold in Steps 2 and 3. Step 1 is correct.

Step 2 and 3): We first insert k elements into H, and then subsequently extract an element from H every time before we insert another one. Therefore we will never have more than k elements, and no less than $k - 1$ elements, until T is empty. In which case H continues to extract all remaining elements in H. The most important point to prove, however, is why this algorithm successfully sorts a k-close sorted list. The essence of the proof lies in the fact that we are dealing with a *finite* list, and thus must have a beginning (and end), that limit the number of potential choices for each index. We prove this through induction on the position j.

For the base case, we insert the first k elements into H, and consider $j = 0$. We know, by definition, that each number in T is less than k positions away from its actual place in the sorted order. Therefore, there must be some number for which the proper location is $j = 0$, and it is less than k positions away. We now prove the base case through contradiction. Assume that we incorrectly sort the element into $A[0]$. This means that must not have considered an element that was k-close this position. Since we are at the first index of the array, there can be no numbers to the left, as they are not valid indices. So this number is contained to the right somewhere between the indices $0 \leq j < k$. H accepts k elements, and it therefore contains all possible candidates for $A[0]$. We therefore had a contradiction, as we indeed considered all k-close elements, so it will properly sort into $A[0]$.

Now with the inductive hypothesis we assume that all numbers up to the j^{th} position have been properly sorted, and we need to determine if the $j + 1$ position will be. First, we recognize that since all $0 \leq j$ positions were properly sorted, all numbers in this range will be 1-close. They will therefore not be candidates for the $j + 1$ position, as a number in a position $\leq j$ must be $\geq$ 2-close to be considered. We thus only consider candidates to the right of $j + 1$, and $j + 1$ itself. They are in positions $j+1 \leq p < j+1+k$. Simplifying to get the size of p we get $0 \leq p < k$. We know this range contains all potential candidates for position $j+1$, and is of size k, so it will fit in H. Extracting the minimum element thus properly sorts it into position $j+1$. QED.

Analysis of Runtime
Step 1: First we create A of size n, which takes $O(n)$ time. Then we create H, which takes $O(k)$ time. This step is $O(k+n)$, which simplifies to $O(n)$.

Step 2 and 3: We will need to pop all n elements from the k-close list, which takes $O(n)$ time and insert them into H, which takes $O(n \log k)$ time. We will also extract all n elements from H, every time extracting the minimum, which takes $O(n \log k)$ time. Finally, we also need to insert all elements into the array A, which takes $O(n)$ time. Combining this with the first step we have $O(n+n+n\log k+n\log k+n)$ which simplifies to $O(n \log k)$ time.

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:

   Algorithm to Find the Longest Path in a DAG
   Step 1) Our graph is defined as $G(V, E, L)$, and we keep the vertices and edges in an adjacency list. The vertices $v \in V$ are the keys, and $e \in E$ are the edges. L is a hash table containing all lengths of edges as values, and the keys are in the form of (v_1, v_2). We define a list S that will operate as a stack to store our topological ordering of the graph. It is assumed that we are dealing with a DAG with real-valued weights including negative weights.

   Step 2) Use a DFS to topologically sort $G(V, E, L)$. Do so by pushing all (V, E, L) onto S as they are popped off the implicit stack (this occurs from smallest post-order to largest). S now contains G in order of decreasing post order.

   Step 3) We present the modified Djikstra below. We pass in S, so the vertices will be iterated through in the topological order from above. The modifications are in bold.

```
DAG Longest Path ($S = (V, E, L); s = \text{Head}(V)$)
    $v, w$: vertices
    dist: array[$V$] of integer
    prev: array[$V$] of vertices
    $H$: priority heap of $V$
    $H := s : 0$
    for $v \in V$ do
        **dist[$v$] := $-\infty$**
        prev[$v$] := nil
    rof

    dist[$s$] := 0
```

```
for $v \in V$ (This will be in order of the topological sort)
    for $(v, w) \in E$
        **if dist**$[w] <$ **dist**$[v] + L(v, w)$
            dist$[w]$ :=dist$[v] + L(v, w)$
            prev$[w]$ := $v$
        fi
    rof
end while, **return (prev, dist)**, end function
```

Step 4) Create an array R to store results. Run the function DAG Longest Path, and pass in S as as input. Use the outputted prev array to determine the longest path, and use the dist array to sum the respective distances for each vertex along the way. Store (prev, d) in R, where d is the summed distance. Pop the head off S, and repeat step 4 until S is empty.

Step 5) Find the longest d in the array, which will yield your longest path: prev$_d$.

Proof of Correctness
Firstly, we know a topological sort works by ordering vertices in decreasing post-order - proved in lecture notes. Our modification of Djikstra works because we simply *invert* it's functionality. Instead of storing minimums, we are storing maximums. The comparison remains the same, and by initializing all vertices to $-\infty$, and deleting maximums, we know that the rest of Djikstra will hold.

The most important optimization is that topological sorting has all the edges pointing in one direction. That is to say the vertices are linearized. We need to show that this allows us to get rid of the heap, which drastically increases runtime. If we update all outgoing edges from v before moving to w, we are guaranteed to have found the shortest path out of v. Since all edges go in one direction, we will never need to reconsider shortest paths. So we need only check each $v \in V$ once, and update all of it's outgoing edges.

Doing so will find the maximum path from a single source node. However, to find all maxes, we need to consider all paths, even if they are in the middle of the graph. To do this, we have to run the algorithm on all V vertices, and store all distances found for these paths. While we could just change the source node, because the topological sort only points in one direction, we opt to remove this in our implementation. Doing this will give us all possible path lengths in the graph. QED.

Analysis of Runtime
First, we need to create our hash table and L and stack S, which will take time $O(V)$. Then we topologically sort the graph, which takes time $O(V + E)$, because it is just a DFS. Insertion into our stack takes $O(V)$ for each vertex. Currently, we are at $O(3V + E) = O(V + E)$. Since we want to find the longest path for *any* given source, we run our algorithm on every vertex (hence why why pop the source after every iteration), to get all possible paths. This modification of Djikstra, however, will run in *linear time* of $O(V + E)$ instead of $O(+E)$, because we know that there is no need for a priority queue or heap. Given our approach we will need to run our algorithm for all V nodes as a source as stated above. This means That overall our runtime will be $O(V^2 + VE)$. While we could do likely do better with memoization, or Dynamic Programming, this implementation is better than most which are $O(V^3)$ time.

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:

   How Buffy will Traverse Streets
   First we need to make some assumptions about Sunnydale. 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". We can show this by contradiction. Assume that Sunnydale is not a connected component, and exclude the case of single, disjoint vertices because they have no representation in this phsyical world (ie. they represent intersections or road ends, but there needs to be a road in the first place). This means that there exists a vertex m, and an edge (u, v) such that if you start at m there is no path to (u, v). This is a contradiction to the layout of Sunnydale, however, because Buffy can traverse the whole city on foot, so there must exist a path between any two vertices.

   Now that we know we are dealing with an undirected, connected component, we can describe how Buffy will achieve this. Buffy will do the following: she will start at any given point and wander. When given the opportunity (an intersection), Buffy will always choose to walk down a road she has not traversed. If she ever ends up at an intersection where the only options are roads she has traversed, or at a dead end, she will turn around and follow the exact path she came down (not just the road, but the path). Remember, if she is given the opportunity to traverse a new road at any moment, she will take it. When Buffy is back at her starting node (intersection or dead end), if she has no new paths she can walk from here, she has traversed the entire city in the manner Giles laid out.

   One way this could be implemented is with a modified DFS that marks edges instead of vertices. Thus we use an adjacency matrix implementation (call it A) to run our DFS. All edges are marked with a 1 in the adjacency matrix. We run the DFS as normal, but after crossing an edge, we change its marking to 2 in A. So instead of traversing to unmarked vertices, we traverse through unmarked edges. After we run the DFS, we follow the path starting in increasing order from the pre and postorder markings.

   Proof of Correctness
   We know this is correct because Sunnydale is a connected component. Thus, we will be able to traverse all nodes of Sunnydale in one DFS, because there are no cousin edges. This is indicated in lecture notes 3, which states that we need to start a new DFS for every connected component. We know that for every node (v, w) there exists a path between them, and that path is must be composed of edges. The post order of the DFS will mimic Buffy's behavior described above: edges are popped off the implicit stack in the order they were placed on them until another open path opens up. Since Sunnydale is a connected component, all (i, j) pre-post order tuples will be contained within the range of the source node tuple (that is to say that there are no disjoint tuples, or that initial node will be the first to be visited, and the last to be popped off the stack). While a DFS doesn't actually "backtrack", Buffy will need to because she is a human. She can achieve this by following

the pre-post ordering as indicated.

Analysis of Runtime
We know that a DFS on an adjacency matrix will take $O(V^2)$ time, but we are marking edges, so this will actually take $O(E^2)$ time. If we store our edge pre and post ordering in an array as we go along, this will take linear time of $O(E)$. Traversing this ordering will also take time $O(E)$. So our total run time will still be $O(E^2)$.

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

Modified Shortest Paths Algorithm
We will use the same implementation of Djikstra presented in page 4 of the Lecture 4 notes. Rather than presenting this algorithm in steps as I have done before, we will provide "edits" to Djikstra, as it asks for a modification. First, instead of a priority heap we will use an array of size $|V-1|m$, which we call A, where each element is a linked list. The two primary changes will be the functionality of deletemin and insert. We will also initialize a counter $i = 0$ at the beginning of the function, which we will use to index our array, and a set S, to keep track of visited vertices, and check if they have been visited in constant time.

Insert Function

The insert function accepts the same arguments as the one in the lecture notes. The modification, however, is that we need insert to operate in constant time of $O(1)$ for each edge traversal to achieve a run time of $O(|E|)$. We will do this by inserting into A at index dist$[w]$ (this is the newly calculated shortest path). This will be constant time because insertion into a linked list is constant time. Before inserting, the function will check to see if the vertex being inserted is in S, and if it is, it will not insert it (because that means it has been inserted before).

DeleteMin Function

The deletemin function accepts the same arguments as the one in the lecture notes. The modification, however, is that we need to delete the minimum in time of $O(m)$ to achieve a run time of $O(m|V|)$ for all vertices. Remember we initialized counter i above. When deletemin is called, we iterate through the array starting from i, checking every $A[i]$. If the $A[i]$ we are on is an empty linked list, increment i by 1 and repeat. If $A[i]$ is not empty, pop the head off this linked list, insert the vertex into S to keep track of it as being visited, assign it as your new minimum, and run the while loop in Djikstra again. If $i = |V-1|m - 1$, and the list at this index is empty, you are finished - return the distance path arrays and end the function.

Proof of Correctness
We know that the only modification to Djikstra is the data structure, so other than this, the algorithm will work as expected. Points to prove are why each vertex will be inserted once into the Array, why we only need an array of size $|V|m$, and why each deletion takes no more than $O(m)$.

Firstly, we know each vertex will be inserted once into the array because we have a set S that stores visited vertices (sets can check for membership in constant time), and we always check against this before insertion.

Second, we only need an array of size $|V-1|m$, because m is the max length. Thus, the worst case graph would have a shortest path that is a Hamiltonian path (traverses all vertices) and the length of each edge between them is m. So a path can be no longer than this. In calculating runtime we obviously drop the -1 because it is negligible.

Finally, the least intuitive, why does each deletion take no more than $O(m)$ if we have an array of size $|V|m$? We know that m describes that maximum distance of any given edge in the graph. Since Djikstra compiles distances as it traverses the graph, we know that from a vertex with distance d, the maximum distance that the next vertex w that will be inserted into A will be $d+m$. We know that if the neighboring vertex has a distance $> d+m$, then it will be ignored by Djikstra as it is not a shortest path candidate. Thus, our iterator will be set at $i = d$, and in the case of a maximum edge, we won't need to iterate any more than m steps. QED.

Runtime Analysis
This algorithm will run in $O(|E| + |V|m)$ as indicated. First, we create an array of size $|V|m$, which takes $O(|V|m)$ time. Then we begin the modified Djikstra function. Every time a new shortest path is found, we need to insert the vertex at the end of that path in the array. Insertion into the array takes constant time, because it takes constant time to calculate the distance of the node (and we use distance as the index), and insertion into a linked list is also constant time. Thus, if we have E edges, we know an invariant is that Djikstra will only look at each edge once, because it uses a BFS technique of looking at all edges emanating from a node, and records the shortest path. In the worst case that we must look at every single edge, insertion will take $O(|E|)$. Thus adding up our current runtimes, we have $(2|E| + |V|m) = |E| + |V|m)$, which includes insertion, checking the set to see if the vertex has already been in the array, and creation of the array.

Deletion, on the other hand, will take $O(m)$ for every vertex. We explained why it will take no more than this in the "Proof of Correctness"" section. Djikstra will have to update the distance for every vertex, and our modified version will never put a visited node back onto the heap. Thus it will take in the worst case (a graph with all maximum edges) m steps for all $V = O(|V|m)$. Adding this to the insertion runtime we get $(|E| + 2|V|m) = (|E| + |V|m)$, as we expected.

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 $2^{\lceil \log_2 n \rceil}$?

Solution:

Solving the Recurrence
We begin by using the recursion-tree method taught in Chapter 4 of CLRS to build a reasonable

guess. As they do, we disregard ceilings and floors initially to make our calculation more reasonable, and then reintroduce them at the end. We observe that we have have the term $n-1$ at the end of our recurrence, and applying a constant that we may need to solve for later. Therefore, $cn-1$ represents the cost of the recursion at the top level. We iterate down a few levels and get the following results, which generalize:

$$cn - 1$$

$$2(\frac{1}{2}cn - 2) = cn - 2$$

$$4(\frac{1}{4}cn - 4) - cn - 4$$

$$= cn - 2^k$$

Because the subproblem sizes continually decrease by a factor of 2 each time, we observe that the depth of the recursion tree is characterized by $\frac{n}{2^k}$. Solving for the base case we get:

$$\frac{n}{2^k} = 1$$

$$\log_2 n = k$$

This means that it takes $\log_2 n$ steps to get to the base level of the tree. So at T(1) we will have $n = 2^k$ nodes, which each have cost T(1). We also have another base case, T(2), which has a value of 1, so we should add this to our calculation for the base level of the tree. We can plug k, T(1), and add 1 into our equation to get the cost for the kth level of the tree:

$$= -2^{\log_2 n} + cnT(1) + 1$$

Finally, we know that we have $n = 2^k = 2^{\log_2 n} = n^{\log_2 2} = n$ nodes at the bottom level as stated above, and the tree takes $k = \log_2 n$ steps to get down to this level. Multiplying gives us a total number of $n \log_2 n$ operations, which represents our primary term. So our guess for the solution is:

$$T(n) = n \log_2 n - 2^{\log_2 n} + cnT(1) + 1$$

Remember that we disregarded ceilings and floors. Since the solution of $T(n)$ represents an upper bound given constant c, we will upper bound the logarithms by using only ceilings. We also simplify for $T(1) = 0$:

$$T(n) = n\lceil \log_2 n \rceil - 2^{\lceil \log_2 n \rceil} + n + 1$$

Proof of Lemmas
Before proving correctness, we will need to prove two separate lemmas that we will use in our induction proof.

Lemma 1:

$$\left\lceil \frac{n}{2} \right\rceil + \left\lfloor \frac{n}{2} \right\rfloor = n$$

We prove the above for $n \in \mathbb{N}$. If n is even, we can describe it as $n = 2m$, and if it is odd we can describe it as $n = 2m + 1$, where m is obviously $\in \mathbb{N}$. For the even case:

$$\left\lceil \frac{2m}{2} \right\rceil + \left\lfloor \frac{2m}{2} \right\rfloor = n$$

$$\lceil m \rceil + \lfloor m \rfloor = n$$

Since m is $\mathbb{N}$, we can remove the floors and ceilings (they do nothing to whole numbers), and complete the even case:

$$m + m = 2m = n$$

For the odd case:

$$\left\lceil \frac{2m+1}{2} \right\rceil + \left\lfloor \frac{2m+1}{2} \right\rfloor = n$$

$$\left\lceil m + \frac{1}{2} \right\rceil + \left\lfloor m + \frac{1}{2} \right\rfloor = n$$

Again, we can remove the floors and ceilings from m:

$$m + \left\lceil \frac{1}{2} \right\rceil + m + \left\lfloor \frac{1}{2} \right\rfloor = n$$

$$2m + 1 = n$$

We have thus proved that $\lceil \frac{n}{2} \rceil + \lfloor \frac{n}{2} \rfloor = n$ for $n \in \mathbb{N}$. QED.

Lemma 2: Let $x, y \in \mathbb{N}$, and $x = \lceil \frac{n}{2} \rceil$ and $y = \lfloor \frac{n}{2} \rfloor$, where y is not a power of 2.

$$\lceil \log_2 x \rceil = \lceil \log_2 y \rceil = \lceil \log_2 n \rceil - 1$$

We will need to prove the following three cases, which cover all of $\mathbb{N}$:

(a) For the first case, we have that both x and y are powers of 2. Let $x = 2^k$ and $y = 2^k$, where $k, j \in \mathbb{N}$. Thus substituting:

$$x = \left\lceil \frac{2^{k+1}}{2} \right\rceil$$

$$y = \left\lfloor \frac{2^{k+1}}{2} \right\rfloor$$

and now simplifying, we can drop the ceilings and floors because if we have a power of two, $\log_2$ must yield a whole number:

$$\left\lceil \log_2 2^k \right\rceil = \left\lceil \log_2 2^k \right\rceil = \left\lceil \log_2 2^{k+1} \right\rceil - 1$$

$$k = k = k + 1 - 1$$

$$= k$$

(b) For the second case, only x is a power of 2. Therefore, we know that n can be described as $2^k - c$, where $0 < c < 2$. This is the only way that x can be a power of 2 with the ceiling, and y will not be. Thus we similarly substitute:

$$x = \left\lceil \frac{2^k}{2} - \frac{c}{2} \right\rceil$$

$$y = \left\lfloor \frac{2^k}{2} - \frac{c}{2} \right\rfloor$$

$$x = \left\lceil 2^{k-1} - \frac{c}{2} \right\rceil$$

$$y = \left\lfloor 2^{k-1} - \frac{c}{2} \right\rfloor$$

Thus we have $0 < \frac{c}{2} < 1$. This thus implies that x and y must be 1 between each other:

$$x = 2^{k-1}$$

$$y = 2^{k-1} - 1$$

$$\left\lceil \log_2 2^{k-1} \right\rceil = \left\lceil \log_2(2^{k-1} - 1) \right\rceil = \lceil \log_2 n \rceil - 1$$

We know that y is not power of 2 by the range of c, thus after applying $\log_2$ to y, it must be greater than $k-2$, which means the ceiling will push it up to $k-1$. Thus:

$$\left\lceil \log_2 2^{k-1} \right\rceil = \left\lceil \log_2(2^{k-1} - 1) \right\rceil = \left\lceil \log_2(2^k - c) \right\rceil - 1$$

$$k - 1 = k - 1 = \left\lceil \log_2(2^k - c) \right\rceil - 1$$

Finally, we also know since $c < 2$, n will be $> k - 1$ so after logging, the ceiling will push it up to k:

$$k - 1 = k - 1 = k - 1$$

(c) For the final case, we assume neither x and y are powers of 2. This therefore means that if n is even $x \leq y$, and if odd, then $x \leq y + 1$. We apply Lemma 1 to get this. Thus a power of 2 must not occur between them, and so

$$\lceil \log_2 x \rceil = \lceil \log_2 y \rceil$$

By Lemma 1, we know $x + y = n$, and neither is a power of 2 so:

$$n > 2 \times 2^{(\lfloor \log_2 x \rfloor)}$$

$$n > 2^{(\lceil \log_2 x \rceil)}$$

$$\lceil \log_2 n \rceil > \lceil \log_2 x \rceil$$

And therefore we have:

$$\lceil \log_2 n \rceil = \lceil \log_2 x \rceil + 1$$

because we know that:

$$\lceil \log_2 n \rceil < \lceil \log_2 x \rceil + 2$$

This is now proved for all cases. QED.

Proof by Induction

We now need to prove our guess to be correct through induction. We first begin with the base cases: we need to plug in $n = 1 = 2$ into our recurrence and solution, and confirm that they equal 0, and 1 respectively. Plugging in for $n = 1$ into our recurrence:

$$T(1) = T(\lceil 1/2 \rceil) + T(\lfloor 1/2 \rfloor) + 1 - 1$$
$$T(1) = T(\lceil 1/2 \rceil) + T(\lfloor 1/2 \rfloor)$$
$$T(1) = T(1) + T(0)$$

Now plugging $n = 1, 0$ into our closed-form solution:

$$T(1) = (1)\lceil \log_2 1 \rceil - 2^{\lceil \log_2 1 \rceil} + 1$$
$$T(1) = 0 - 1 + 1 = 0$$

Now plugging $n = 2$ into our recurrence and solving with the above:

$$T(2) = T(\lceil 1 \rceil) + T(\lfloor 1 \rfloor) + 2 - 1$$
$$T(2) = T(1) + T(1) + 1$$
$$T(2) = 0 + 0 + 1 = 1$$

Both of the base cases check out. Now we proceed to the inductive step, and will prove this through strong induction. We assume that the recurrence holds for all i where $1 \leq i \leq n$ and we need to prove it for T$(n + 1)$. First, we plug in $n + 1$ into our recurrence:

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

Now we use our solution to substitute in for $T(\lceil (n+1)/2 \rceil)$ and $T(\lfloor (n+1)/2 \rfloor)$, which we can do through strong induction:

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

Now we use Lemma 2 to substitute into 4 places. Remember $n + 1$ is still a natural number, so as long as we keep it in parenthesis, we can treat the lemmas the same:

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

Simplifying terms:

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

Now we can pull common terms so we can apply Lemma 1:

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

Plugging $n+1$ into our closed-form solution gives us the exact same as the above, so we have proven this through induction. QED.

Collaborators

Matt Neary
Conor Quinn
SJ Kim