CS120: Intro. to Algorithms and their Limitations Prof. Salil Vadhan

Problem Set 4

Harvard SEAS - Fall 2021 *Due: Wed Oct. 12, 2022 (11:59pm)*

Your name: Nicole Chen
Collaborators: Jayden Personatt
No. of late days used on previous psets: 0
No. of late days used after including this pset: 3

1. (Randomized Algorithms in Practice)

   (a) Implement Randomized QuickSelect, filling in the template we have given you in the Github repository.

   (b) In the repository, we have given you datasets x_n of key-value pairs of varying sizes to experiment with. For each dataset x_n and any given number k, you will compare two ways of answering the k selection queries Select($x_n, [n/k]$), Select($x_n, [2n/k]$), . . ., Select($x_n, [(k-1)n/k]$) on x_n, where $[\cdot]$ denotes rounding to the nearest integer:

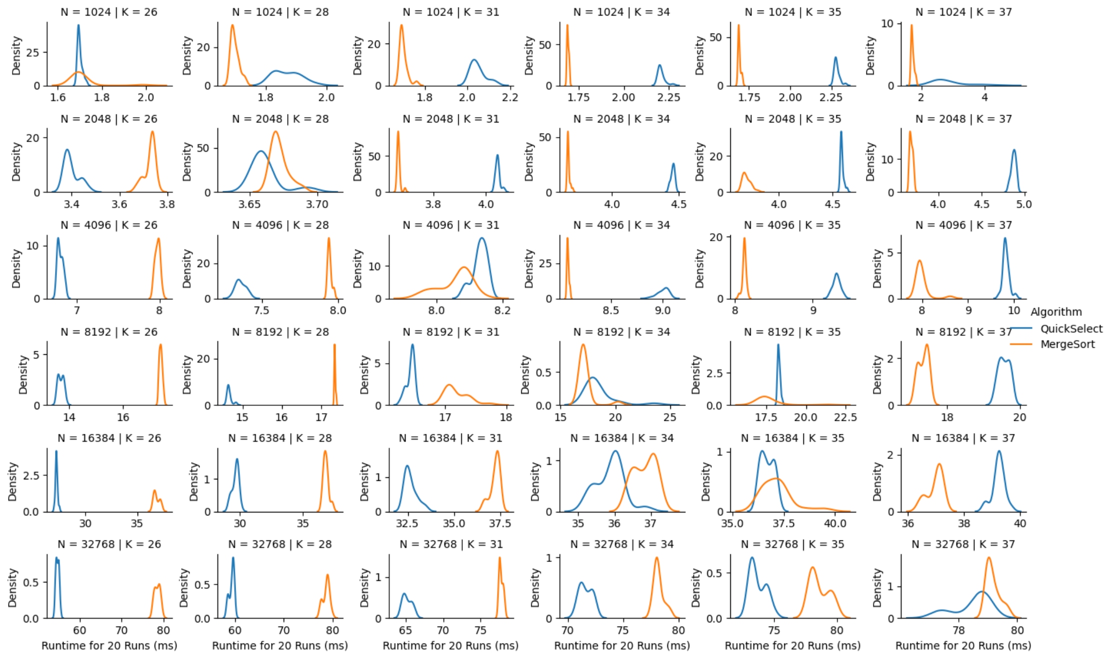
      i. Running Randomized QuickSelect k times

      ii. Running MergeSort (provided in the repository) once and using the sorted array to answer the k queries

   Specifically, you will compare the *distribution* of runtimes of the two approaches for a given pair (n, k) by running each approach many times and creating density plots of the runtimes. The runtimes will vary because Randomized QuickSelect is randomized, and because of variance in the execution environment (e.g. what other processes are running on your computer during each execution).

   We have provided you with the code for plotting. Before plotting, you will need to implement MergeSortSelect, which extends MergeSort to answer k queries. Your goal is to use these experiments and the resulting density plots to propose a value for k, denoted k_n^*, at which you should switch over from Randomized QuickSelect to MergeSort for each given value of n. Do this by experimenting with the parameters for k (code is included to generate the appropriate queries once the ks are provided) and generate a plot for each experiment. Explain the rationale behind your choices, and submit a few density plots for each value of n to support your reasoning. (There is not one right answer, and it may depend on your particular implementation of QuickSelect.)

   (Solution) Below is a table of different k_n^* and n pairs that denotes for each n, what k would allow MergeSort's algorithm to run on par with QuickSelect's. In other words, when should we switch from the QuickSelect algorithm to MergeSort. This was produced through analyzing the overlap between QuickSelect's and MergeSort's density plot graphs. More specifically, I wanted to identify overlaps between the peaks of each sort's graphs, which would signify that at n, MergeSort and QuickSelect have similar run times for those k queries. Because for smaller values of k, QuickSelect consistently outperformed MergeSort in run time, the decrease in the difference in run time amongst the density plots as k increased shows how MergeSort slowly optimizes run time for large k. As a result, when both density plots overlapped, that signified the point where MergeSort's run time at k queries on a list of size n would “overtake” QuickSelect's, and therefore, it would be optimal to switch algorithms. If we look at $n = 2048$ and $k = 28$ in the graph below, we can see that at smaller values of k where $k < 28$, the peaks for QuickSelect are to the left of the peaks for MergeSort. This signifies that QuickSelect outperforms MergeSort in run time at $k < 28$ queries. However, if we move across the graph, we see that at $k = 28$, there is significant overlap between MergeSort's and QuickSelect's run times, and at $k = 31 > 28$, MergeSort becomes significantly faster as its peaks are to the left of QuickSelect's. Thus, the optimal k to switch from QuickSelect to MergeSort for a size of $n = 2048$ is at $k = 28$. We can extend this reasoning to the other values of n. As we increase n, the threshold for when to switch also increases, which makes sense because it would take MergeSort longer to sort the list and therefore more queries would need to be called to make up for the longer preprocess time. One thing I did want to note is how the density plot distribution for QuickSelect varied and encompassed a far wider width than MergeSort. This can be attributed to the fact that because our QuickSelect is a Las Vegas Algorithm, there is far more variation in the run time because it is dependent on the random pointer selected.



n	k_n^*
1024	26
2048	28
4096	31
8192	34
16384	35
32768	37

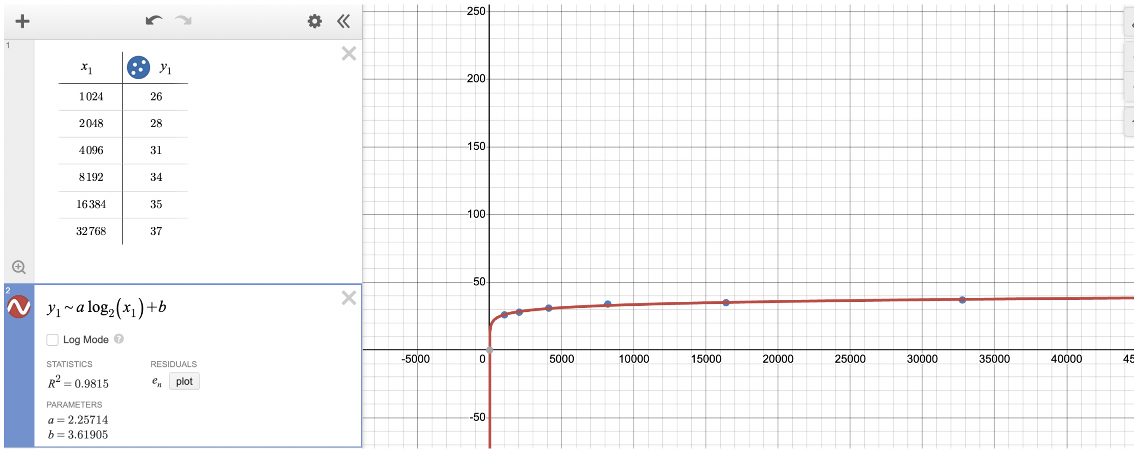
(c) Extrapolate to come up with a simple functional form for k_n^*, e.g. something like $k^*(n) = 3\sqrt{n} + 6$ or $k^*(n) = 10 \log n$. (Again there is not one right answer.)

(Solution) To extrapolate a simple functional form for k_n^*, I plotted the points in the above table in Desmos to create a scatter plot. To identify the best type of function-form to fit the scatter plot, I compared the worst case run time for MergeSortSelect and QuickSelect of k_n^* queries, setting both run times equal to reach other because k_n^* is the point where both worst case run times should be equal. MergeSortSelect's worst-case run time is $O(n\log(n) + k_n^*)$ because the sort take $O(n\log(n))$ and the subsequent k_n^* queries takes $O(1)$ time each. QuickSelect's worst-case run time is $O(k_n^*(n))$ because each query selection takes $O(n)$ time.

$$\begin{aligned} n\log(n) + k_n^* &\approx k_n^*(n) \\ n\log(n) &\approx k_n^*(n-1) \\ \frac{n\log(n)}{n-1} &\approx k_n^* \\ \log(n) &\approx k_n^* \end{aligned} \tag{1}$$

Therefore, our function-type should resemble a log function (of base 2) as above, with variations in the constants multiplied and added to that function form. Below is a picture showing the scatterplot and the best fit function.In summary, we extrapolate to come up with a simple functional form of k_n^* according to Desmos's best fit.

$$k_n^* \approx 2.26\log(n) + 3.62 \tag{2}$$



x_1	y_1
1024	26
2048	28
4096	31
8192	34
16384	35
32768	37

$y_1 \sim a\log_2(x_1) + b$

Log Mode

STATISTICS

$R^2 = 0.9815$

RESIDUALS

e_n plot

PARAMETERS

$a = 2.25714$

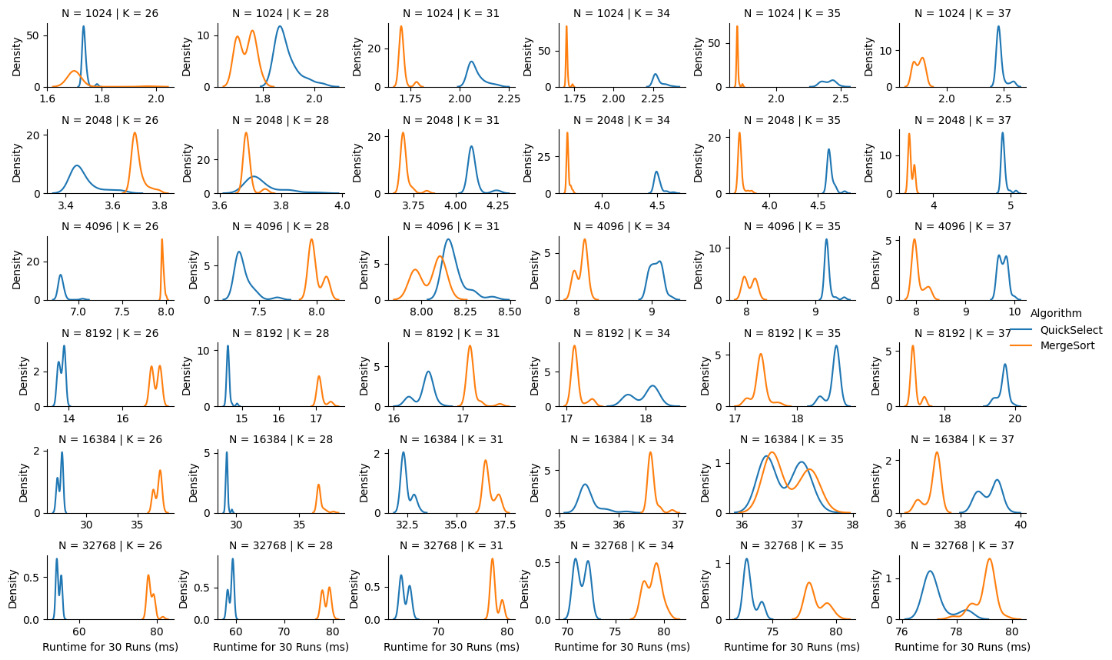
$b = 3.61905$

(d) (*optional) One way to improve Randomized QuickSelect is to choose a pivot more carefully than by picking a uniformly random element from the array. A possible approach is to use the ***median-of-3*** method: choose the pivot as the median of a set of 3 elements randomly selected from the array. Add Median-of-3 QuickSelect to the experimental comparisons you performed above and interpret the results.

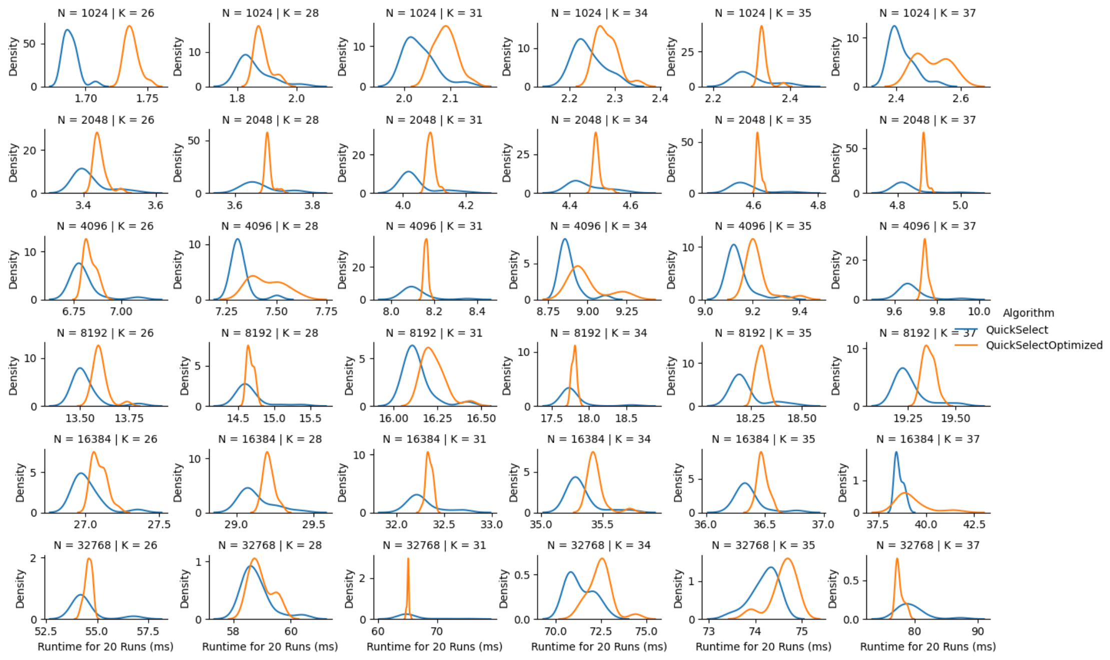
(Solution) For this section, I added 6-8 lines of code to implement the ***median-of-3*** method. Namely, I created a list of pointers that has max length 3, ensuring that each of those 3 elements in pointers is unique. Then, I found the medium of those 3 pointers and used that as my pivot. If the inputted list had less than 3 elements, then the pivot was just a random pivot.

```
pointers = []
if len(arr) > 3:
    while len(pointers) < 3:
        pointers.append(get_random_index(arr))
    pointer = statistics.median(pointers)
else:
    pointer = get_random_index(arr)
p = arr[pointer][0]
```

To compare the newly implemented ***median-of-3*** method with MergeSort, I graphed the distribution plot for ***median-of-3*** against MergeSort. The below showcases this comparison with the same values of k_n^* that I had for QuickSelect before the improvement. Please note: the blue is ***median-of-3*** and the orange is MergeSort.



I noticed though that the distributions were extremely similar to the distributions between regular QuickSelect and MergeSort. At $n = 1024$, ***median-of-3*** still had significant overlap with MergeSort at $k = 26$. This was the same across most n's and signified that ***median-of-3*** has little impact on improving the overall distributed run time of QuickSelect when compared to MergeSort. This can be attributed to the fact that even though our pointer may be more "precise" and representative of the input, our fundamental algorithm does not change. I subsequently decided to make a more direct comparison between the original QuickSelect algorithm and ***median-of-3*** by comparing the run time distributions of both to each other. This was because I hypothesized that by selecting the medium pointer of 3 randomly selected indexes, our run times distributions for ***median-of-3*** would be more clustered/focused rather than widely distributed around the actual medium of the array. By nature of selecting the medium out of 3 elements, ***median-of-3*** would allow us to narrow down our pointers to a smaller set of values that would in general be closer to the medium of the input array. This is validated by the graph below, where across k, ***median-of-3*** (denoted by orange) had run time distributions that were more clustered than that of the original QuickSelect. The center for each distribution across all k's and n's between QuickSelect and ***median-of-3*** seemed roughly the same (did not overlap).



2. (Dictionaries and Hash Tables) Recall the DuplicateSearch problem from Lecture 3. Show that DuplicateSearch can be solved by a Las Vegas algorithm with expected runtime $O(n)$ using a dictionary data structure. (You can quote the runtimes of the implementation of a dictionary data structure from Lecture 9 without proof.)

   (Solution) We first show that our algorithm `DuplicateSearchHash` is a Las Vegas algorithm that correctly solves the Duplicate Search problem.

   (a) Choose a random hash function $h : [U] \to [m]$ such that for each element in the inputted data set B with elements $b_0 ... b_{n-1}$ that are key value pairs, where $i \in 0, ...n-1$, such that the keys $K_i \in [U]$. We then initialize an array A of size $m << U$, which can be considered our dictionary data structure (hash table) that maps each key-value pair (K_i, V_i) uniquely to an index in the hash table between 0 and m. We set $m = n$ and assume that there exists a hash function $h : [U] \to [m]$ that maps each unique value in the inputted data set to a unique $h(K_i)$ such that for n data set elements, there are $n = m$ indices in the array A which guarantees no collisions should the values of B be unique.

(b) We store the random hash function h to be able to access it as we iterate across all n elements in array A.

(c) For all (K_i, V_i) pairs in A such that $K_i \in U$, we evaluate $h(K_i)$ and append the value of K to a linked list at the index $h(K_i)$ such that $A[h(K_i)]$ is a linked list that stores the key value pair (K_i, V_i).

(d) Iterate through the array B of inputed elements, applying the hash function $h(K_i)$ to every key value pair (K_i, V_i) in A to essentially "search" for that key value pair in A.

(e) For each hash function value $h(K_i)$, check to see if there are more than two elements in each linked list at $A[h(K_i)]$ If there are more than two elements, then you've found a duplicate. Return the duplicate.

(f) If you have iterated through all key value pairs and have not yet returned a duplicate, return $\perp$.

We now explain how the above algorithm successfully finds the duplicate value and is a Las Vegas Algorithm. Namely, we have a random hash function h such that every element in the inputted data set B is mapped to a unique index between $0...m-1$. This is possible because we set $m = n$ and therefore, n elements should be slotted into $m = n$ slots in our hash table without collisions if there are no duplicates. Because only duplicate values in the dataset B receive the same hashed value $h(K_i)$, when we insert each key-value pair to one of the m indices in our hash table, we expect that only duplicate values will be inputted into the same slot/index. Therefore, by iterating through our hash table and checking at each index (a linked list) if there are multiple values associated with that index, we can discover whether there are duplicate values in our set or not. If there are duplicate values, then $A[h(K_i)]$ will have a size greater than 1. If there are no duplicates, we will have iterated through all indices of A where $A[h(K_i)]$ has size 1 for all i, and therefore return $\perp$. The fact that our algorithm utilizes a *random* hash function and always returns the duplicate or $\perp$ when there is no duplicate, makes our algorithm a Las Vegas Algorithm.

`DuplicateSearchHash` has an expected run time of $O(n)$. We proved in class that the dictionary data structure (hash table) is a Las Vegas data structure that takes $O(1)$ to generate a hash function, $O(1)$ space to store the hash function, and $O(1)$ time to evaluate $h(x)$ for all $x \in U$. Creating our hash table A takes $O(m) = O(n)$ time because our hash table has a size of $m = n$. Therefore, in total, steps (a), (b), and (c) takes an expected run time of $O(n)$ as the n elements we iterate through to hash and populate into our hash table A of size $m = n$ each takes $O(1)$ time to make $O(n)$ time for all n. Step (d) also takes $O(n)$ because we need to iterate through all n values of the array A and apply the hash function to each element, which takes $O(1)$ time. For each of the elements we hash in (e), we access the array A at index $h(K_i)$ in constant time and "iterate" through the linked list in the sense that after the first element in $A'[h(K_i)]$ we check to see if there is a second element. This has an expected run time of $O(1)$ because we first check to see if there is one element in the linked list at $A'[h(K_i)]$. After checking the first element, we check if there is a second element. If there is, then we know our inputted data set has a duplicate and can just return that duplicate, terminating our program. This takes two constant run-time steps: hence, $O(1)$. Because (e) is embedded within step (d), iterating through array A to get $h(K_i)$ and subsequently checking for duplicates within each linked list in $A'[h(K_i)]$ takes a total of $O(n)$ time. Returning either the duplicate or $\perp$ in (f) takes constant time. Therefore, we have a total run time of $O(n) + O(n) + O(1) = O(n)$.

3. (Rotating Walks) Suppose we are given k digraphs on the same vertex set, $G_0 = (V, E_0), G_1 = (V, E_1), \dots, G_{k-1} = (V, E_{k-1})$. For vertices $s, t \in V$, a *rotating walk* with respect to $G_0, \dots, G_{k-1}$ from s to t is a sequence of vertices $v_0, v_1, \dots, v_\ell$ such that $v_0 = s$, $v_\ell = t$, and $(v_i, v_{i+1}) \in E_{i \bmod k}$ for $i = 0, \dots, \ell - 1$. That is, we are looking for walks that rotate between the digraphs $G_0, G_1, \dots, G_{k-1}$ in the edges used.

   (a) Show that the problem of finding a Shortest Rotating Walk from s to t with respect to $G_0, \dots, G_{k-1}$ can be reduced to Single-Source Shortest Walks via a reduction that makes one oracle call on a digraph G' with kn vertices and $m_0 + m_1 + \cdots + m_{k-1}$ edges, where $n = |V|$ and $m_i = |E_i|$. We encourage you to index the vertices of G' by pairs (v, j) where $v \in V$ and $j \in [k]$. Analyze the running time of your reduction and deduce that the Shortest Rotating Walk can be found in time $O(kn + m_0 + \cdots + m_{k-1})$. To test your reduction and algorithm, try running through the example in Part 3b.

   (Solution) We first show our reduction from Shortest Rotating Walk to Single Source Shortest Walks.

      i. We first create our digraph G' through iterating through each vertex set (V) of each digraph $G_0..G_{k-1}$ k times to create k different sets of vertices where each vertex is (v, j) with $v \in V$ and $j \in [k]$. Note: the j corresponds to the "label:" G_j. This creates kn vertices.

      ii. Through a nested loop where we traverse each edge starting from digraphs G_0 to G_{k-1}, we create the edges of G' such that the edge (v, v') where $v, v' \in V$ in a digraph becomes $((v, j), (v', j + 1 \mod k))$. In other words, the first vertex in the edge becomes (v, j) instead of v and the second vertex in the edge becomes $(v', j + 1 \mod k)$ instead of v'. This is a nested loop because we need to access E_i of each digraph G_i and iterate across E_i to update our edges to the form $((v, j), (v', j + 1 \mod k))$. We will have therefore created $m_0 + ... m_{k-1}$ edges in G' because we iterate through each edge in edge set of the k digraphs.

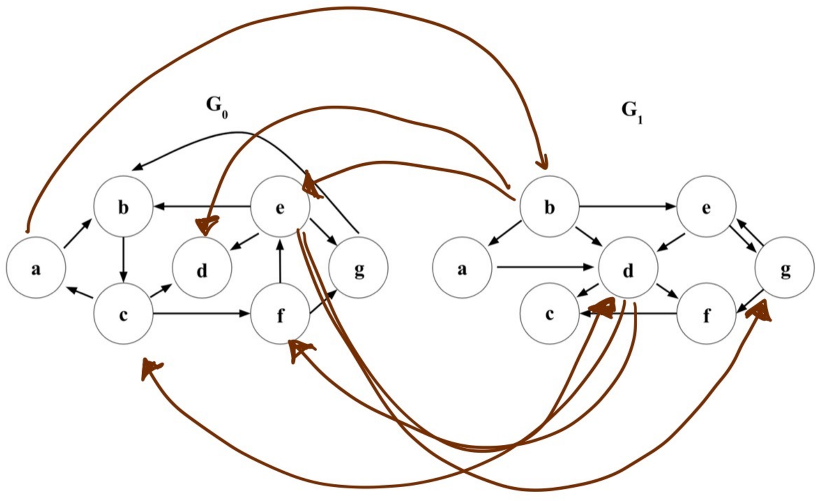
      iii. With the digraph G', we can make an oracle call of the Single-Source Shortest Walks algorithm on G'. An example of such an algorithm is Breadth First Search.

      iv. If the value returned from the oracle call is ∞ then return $\perp$. Otherwise, return the path p_v that offers the shortest distance from s to t.

   We now show that our above algorithm correctly finds the shortest rotating walk from s to t. In creating k different sets of vertices of the form (v, j), we ensure that each vertex is accounted for across all k digraphs. Subsequently, by iterating through each edge set of all k digraphs and then iterating again through each edge of each edge set to draw the neighbors of the vertex in G_i to its paired neighbor is $G_{i+1 \bmod k}$, we ensure that all possible rotations are accounted for. Therefore, our G' digraph correctly accounts for all vertices and edges amongst digraphs $G_0...G_{k-1}$. A call to Single-Source Shortest Walks on G' will provide the minimized distanced-path s to t. There may be multiple paths as we can start from $(s, 0)$ to any vertex at $(t, 0), ...(t, k - 1)$, but we want to find the shortest distance path, getting rid of the indexes associated at each vertex when we've found this path because we care about s to t only. Our call to Single-Source Shortest Walk will return the path p_v for the shortest distance if t is reachable. If not, then Single Source Shortest Walk will output ∞ and our algorithm would return $\perp$.

We now show that our algorithm has a run time of $O(kn + m_0 + \cdots + m_{k-1})$. In (i), we iterate through each vertex set (V) n times. Each vertex set has a size of n. Thus, creating the vertices of G' requires a run time of $O(kn)$ as each vertex set has n vertices that need to be duplicated k times. In (ii), we utilize a nested loop to traverse the edges of each digraph from G_0 to G_{k-1}. We populate each edge in G' accordingly through modifying the vertices with their new assignments (as listed in (ii)). These new assignments take $O(1)$ time as we are simply doing mathematical operations, but because we have to do this for every edge in each of the k digraphs, each of which have their original $m_i = |E_i|$ number of edges, the run time for populating G''s edges is $O(m_0+m_1+...m_{k-1})$ for all edges in all k digraphs. In (iii), we make an oracle call of the Single-Source Shortest Walks algorithm on G'. If we were just assessing the reduction algorithm's run time, we would consider the oracle a one-time step call, so the third step takes $O(1)$ time. Including the run time of the Single-Source Shortest Walks algorithm though, which we will use the BFS algorithm, we find that because there are kn vertices in G' and $m_0 + m_1 + ...m_{k-1}$ edges in G', based on Theorem 5.1 in Lecture 11 notes, Single Source Shortest Walks has a run time of $O(n+m) = O(kn+m_0+m_1+...m_{k-1})$. We now account for getting rid of the indexes associated with each vertex because we only care about the path from s to t. Worst case for that is $O(kn)$ because we would iterate through all the vertices in G'. Thus, our Shortest Rotating Walk has run time of $O(kn) + O(m_0 + m_1 + ...m_{k-1}) + O(kn + m_0 + m_1 + ...m_{k-1}) + O(kn)$ which can be simplified to $O(kn + m_0 + \cdots + m_{k-1})$.

(b) Run your algorithm from Part 3a on the following pair of graphs G_0 and G_1 to find the Shortest Rotating Walk from $s = a$ to $t = c$; this will involve solving Single-Source Shortest Walks on a digraph G' with $2 \cdot 8 = 16$ vertices. Fill out the table provided below with the BFS frontier in G' at each iteration, labelling the vertices of G' as $(a, 0), (b, 0), \ldots, (g, 0), (a, 1), (b, 1), \ldots, (g, 1)$, and for each vertex v in the table, drawing an arrow in the graph from v's BFS predecessor to v.



d	Frontier F_d	Predecessor Relationships
0	(a, 0)	N/A
1	(b, 1)	(a, 0)
2	(d, 0) (e, 0)	(b, 1)
3	(d, 1) (g, 1)	(e, 0)
4	(c, 0) (f, 0)	(d, 1)

(c) A group of three friends decides to play a new cooperative game (similar to the real-life board game Magic Maze). They rotate turns moving a shared single piece on an $n \times n$ grid. The piece starts in the lower-left corner, and their goal is to get the piece to the upper-right corner in as few turns as possible. Many of the spaces on the grid have visible bombs, so they cannot move their piece to those spaces. Each player is restricted in how they can move the piece. Player 0 can move it like a chess-rook (any number of spaces vertically or horizontally, provided it does not cross any bomb spaces). Player 2 can move it like a chess bishop (any number of spaces diagonally in any direction, provided it does not cross any bomb spaces). Player 3 can move it like a chess knight (move to any non-bomb space that is two steps away in a horizontal direction and one step away in a vertical direction or vice-versa). Using Part 3b, show that given the $n \times n$ game board (i.e., the locations of all the bomb spaces), they can find the quickest solution in time $O(n^3)$. (Hint: give a reduction, mapping the given grid to an appropriate instance $(G_0, G_1, \ldots, G_{k-1}, s, t)$ of Shortest k-Rotating Paths.)

(Solution) We utilize the reduction algorithm Part 3b to show that the quickest solution can be fond in time $O(n^3)$. Namely, we instantiate a digraph G' that resembles the $n \times n$ game board such that there are n^2 vertices. The digraph G' will be composed of smaller digraphs $(G_0, G_1, \ldots, G_{k-1})$ where each of the k smaller digraphs is composed of the same vertex set V with n^2 vertices. There are $k = 3$ smaller digraphs that represent the different types of movements the players can perform - rook, bishop, and knight.

Our digraph G' utilizes a k-rotating path strategy to optimize finding the shortest walk from s (lower-left corner) to t (upper-right corner) through grouping all $k = 3$ digraphs together, rotating amongst the k digraphs to find the best path. We show this algorithm in steps below.

i. We have $k = 3$ digraphs that map the different ways each player can move. More specifically, let G_0 be the digraph with edges that connects either horizontal or vertical pieces on the board. Therefore, the edges of G_0 can only connect vertices that are vertically or horizontally adjacent to one another. Similarly, let G_1 be the digraph that has edges that connect the diagonal pieces on the board. Lastly, let G_2 be the digraph with edges that connect vertices that are either 2 steps horizontally and 1 step vertically away (or vice versa). Another way to think about this is that G_0 are the moves player 1 can make, G_1 are the moves player 2 can make, and G_2 are the moves player 3 can make.

ii. G_0, G_1, G_2 each have a set of vertices V and edges E_i such that $n^2 = |V|$. Because the board is an $n \times n$ board, there are n^2 locations and therefore n^2 vertices in each digraph. It is important to note that edges can only correspond to vertices/places that the player can move. If there is an obstacle, then certain edge relationships cannot be expressed because of that barrier. For G_0, $n^2(2n-2) = |E_0|$ because for each vertex, the maximum number of vertices (when there are no obstacles) a horizontal or vertical motion can move to is $2n-2$. There are n^2 vertices, so the maximum number of edges is $n^2(2n-2) = |E_0|$. For G_1, $n^2(2n-2) = |E_1|$ because for each vertex, the maximum number of vertices (when there are no obstacles) a diagonal motion can move to is $2n-2$. There are n^2 vertices, so the maximum number of edges is $n^2(2n-2) = |E_1|$. Lastly, for G_2, $8n^2 = |E_2|$ because the knight has maximum 8 other vertices it can move to.

iii. We now know the size of V and E_i for each of our $k = 3$ digraphs. We then want to populate G' with appropriate vertices and edges. We do so by referencing Part 3a. Accordingly, we populate G' with vertices (v, j) where $v \in V$ and $j \in [k]$ and edges in G' such that the edge (v, v') becomes $((v, j), (v, j+1 \mod k))$. Doing so takes $O(3n^2 + n^2(2n-2) + n^2(2n-2) + 8n^2)$ as shown in Part 3a, where in our case, $k = 3$, n^2 comes from the number of vertices in our $n \times n$ board, $m_0 = n^2(2n-2)$, $m_1 = n^2(2n-2)$, $m_2 = 8n^2$. We can simplify our run time to $O(n^3)$ as we only care about the highest powers of n for the worst case scenario.

iv. We have instantiated G' and also explained the run time of the reduction to Shortest Rotating Walks($O(n^3)$) which we use in our algorithm. More in-depth logic can be referenced through Part 3b. Another way we can think about this is after instantiating G', we make one oracle call of the algorithm Single-Source Shortest Walks on the digraph G' to find the the shortest walk from s (lower-left corner) to t (upper-right corner). This is essentially the exact process that we did for Shortest Rotating Walks, so a reduction to Shortest Rotating Walks with digraphs G_0, G_1, G_2 would suffice. However, we can continue with the logic of the oracle call to Single-Source Shortest Walks to prove the same point. Let's invoke the Breadth First Search algorithm to do so, where in lecture notes **Theorem 5.1** indicated our BFS algorithm takes $O(n+m)$ time. n (ie. size of digraph) in G' is n^2, m in G' is $n^2(2n-2) + n^2(2n-2) + 8n^2$. Therefore, our Single-Source Shortest Walk BFS

Algorithm discovers the shortest walk from s to t in G' in $O(n^3)$ time. As our reduction algorithm $T(n)$ has a run time of $O(n^3)$ and our subroutine has a run time of $O(n^3)$, we have shown that our algorithm solves the Magic Maze problem in $O(n^3)$ with a reduction to Single-Source Shortest Walk. However, this can be more succinctly done as a reduction to Shortest Rotating Walks, with a run time of $O(n^3)$ as well because we would be feeding in G_0, G_1, G_2 into Shortest Rotating Walks.