**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: Aika Aldayarova**
**Collaborators: Avi Gulati, Liya Jin**
**No. of late days used on previous psets: 1**
**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 $k$s 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:
      To find the value of k at which one should switch from using Randomized QuickSelect to MergeSort, we need to find the value of k at which the density plots of both of the algorithms overlap the most. This would mean that at this value of k, the run-times of the two algorithms are the same, whereas at other values of k, one algorithm will be performing better than the other. By finding a value of k at which the graphs overlap, we will find the inflection point at which it would be safe to switch from one algorithm (that may perform better for smaller datasets) to the other algorithm (that may perform

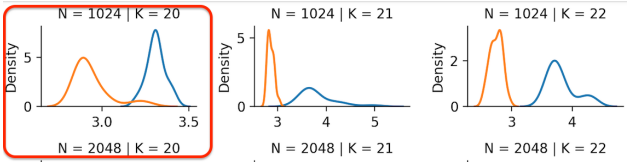
better for larger datasets). With this intuition in mind, the k values I found to be the best-fitting for each value of $n$ are as follows:

| $n$ | Proposed $k_n^*$ |
|---|---|
| 1024 | 20 |
| 2048 | 21 |
| 4096 | 22 |
| 8192 | 23 |
| 16384 | 24 |
| 32768 | 25 |

Below I provide a brief justification for each k value and a few density graphs.

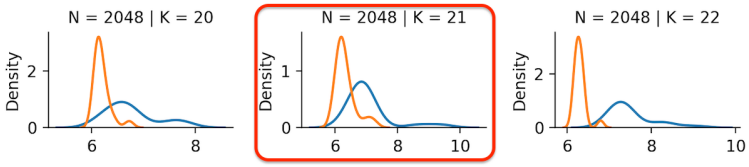
- $\underline{n = 1024}$: $k = 20$ is the best k value because the graphs overlap the most and the x-scale of the graphs is much tighter than in all other density graphs for this n, which means that the data is actually much more aligned than it is displayed to be.

Figure 1: Among the three graphs k=20 is the best for n=1024.



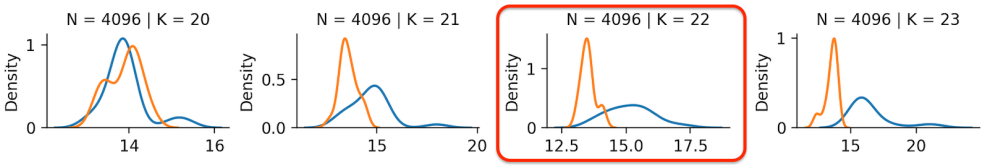

- $\underline{n = 2048}$: $k = 21$ is the best k value because this looks like the point at which the graphs begin to switch positions.

Figure 2: Among the three graphs k=21 is the best for n=2048.



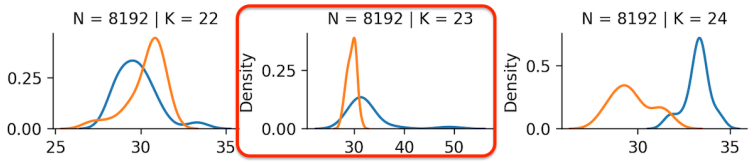

- $\underline{n = 4096}$: $k = 22$ or $k = 21$ seem like they could be great values since values right before and right after them produce graphs that are in exactly opposite sides. However, upon closer inspection, we see that $k = 22$ has a finer x-scale, which means that although its graph looks similar to that of $k = 21$, because it has a finer scale, its graphs are more aligned than that of $k = 21$.

Figure 3: Among the three graphs k=22 is the best for n=4096.



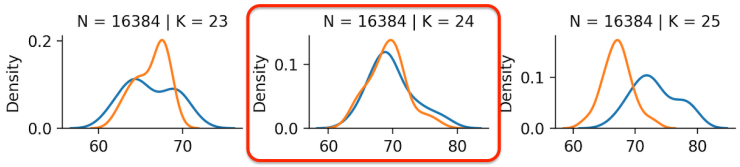

- $\underline{n = 8192:}$ $k = 23$ is the best k value because it shows the inflection of the two graphs at k values right before and after it.

Figure 4: Among the three graphs k=23 is the best for n=8196.



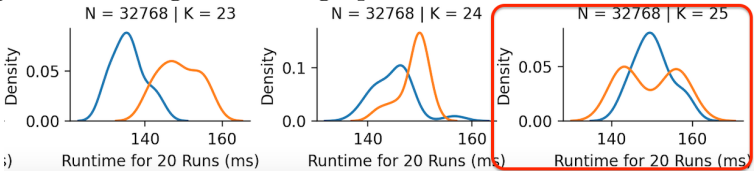

- $\underline{n = 16384:}$ $k = 24$ is the best k value because it shows the inflection of the two graphs at k values right before and after it.

Figure 5: Among the three graphs k=24 is the best for n=16384.




- $\underline{n = 32768:}$ $k = 25$ is the best k value because it seems to overlap the two density graphs the best and seems to indicate the location of the inflection point.

Figure 6: Among the three graphs k=25 is the best for n=32768.




(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.)

One functional form of $k_n^*$ could be $k^*(n) = 2 \log n$. This functional form arithmetically matches the k values from the previous question and this is made explicit in the table

below:

| $n$ | Proposed $k_n^*$ | $2 \log n$ |
|---|---|---|
| 1024 | 20 | 20 |
| 2048 | 21 | 21 |
| 4096 | 22 | 22 |
| 8192 | 23 | 23 |
| 16384 | 24 | 24 |
| 32768 | 25 | 25 |

This functional form also aligns well with the run-times of MergeSort and QuickSelect. We know from lecture 2 notes that MergeSort's runtime is $O(n \log n + k)$ and we know from lecture 8 notes that QuickSelect's runtime is $O(kn)$ where k is some constant. Since we were trying to find a value of k at which one would want to switch from using one algorithm to another given some n, we wanted to find a k that would make the two run-times of the algorithms equal. If we make the functional form of k equal $2 \log n$, then this would make MergeSort's runtime equal $O(n \log n + 2 \log n) = O(n \log n)$ and QuickSelect's run-time equal $O(2n \log n) = O(n \log n)$. So, the run-times of the two algorithms would thus be asymptotically same, and this would mean we found our sought-after k value.

(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.

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.)

Before we analyze the run-time of the DuplicateSearch problem implemented as a Las Vegas algorithm, let's list out the concrete steps of what the implementation would look like:

(a) First, given some input array, we would want to iterate through each element in that array and compute its corresponding location within the hash table.

(b) Then, we would want to navigate to that computed location.

(c) Finally, we would want to check to see if there already exists a copy of that element. If there is, then it is a duplicate and we can return the duplicate element. Else, we can insert the element into the slot within the hash table. We would want to repeat these steps for each element in the input array.

Having outlined the concrete steps of the Las Vegas algorithm implementation, we can now analyze its run-time. The first step would take $O(1)$ time by the definition of a hash function run-time provided in lecture 9 notes. The second step would also take $O(1)$ time since it is a

simple operation. The third step would also take $O(1)$ time to check the duplicate as deduced in lecture 9 notes. Since we'll be doing the following steps for every element in the input array of length $n$, the overall run-time of this algorithm will therefore be $O(n)$.

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

   (a) Show that the problem of finding a Shortest Rotating Walk from $s$ to $t$ with respect to $G_0, \ldots, G_{k-1}$ 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.

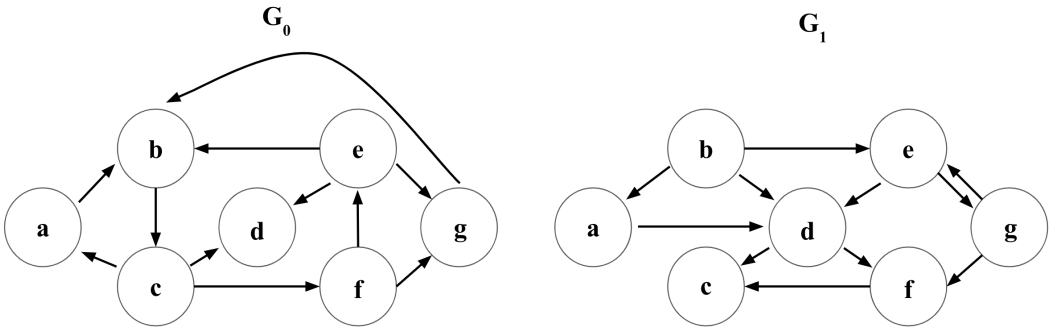
   *Proof.* Let $\Pi$ be a computational problem "Shortest Rotating Walk" and let $\Gamma$ be another computational problem "Single-Source Shortest Walks" which will be a subroutine for $\Pi$. In order to show that $\Pi \leq \Gamma$ making one oracle call on the diagraph $G'$, we can imagine the process working out as follows:

   i. Preprocess: We need to construct $G'$ (per the description), using the input to $\Pi$. This $G'$ will have $kn$ vertices such that $kn$ is the size of $V'$, $n$ is the size of $V$, and $m_0 + m_1 + \cdots + m_{k-1}$ is the size of $E'$ (where each $m_i$ is the size of $E_i$). Together, the preprocessing step should take $O(kn)$ time to make $V'$ and $O(m_0 + m_1 + \cdots + m_{k-1})$ time to make $E'$.
   ii. Oracle Call: Having constructed our $G'$, we make one oracle call to $\Gamma$ using the newly-minted $G'$ and some $s'$ starting vertex as inputs. By definition of the computational problem $\Gamma$, the output (which will be the distance and path from $s$ to all other reachable vertices in $G'$) will take $O(1)$ time to surface.
   iii. Postprocess: Finally, we need to alter the output of the subroutine to match the output format of the original computational problem. $\Gamma$ outputs a series of distances and paths from a staring vertex to all other reachable vertices. $\Pi$ requires its output to be a sequence of vertices that make up the shortest walk. Thus, part of postprocessing will require us to iterate through each of the outputs created in $\Gamma$ and determine which one is the most efficient/shortest. Together, this step should thus take $O(kn)$ time.

   In conclusion, the run-times of each of the three steps above can be added together and the overall run-time of the reduction will be $O(kn + m_0 + \cdots + m_{k-1})$.
   Q.E.D.

   (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) | $\emptyset$ |
| 1 | (b,1) | $(a,0) \to (b,1)$ |
| 2 | (d,0),(e,0) | $(b,1) \to (d,0), (b,1) \to (e,0)$ |
| 3 | (d,1),(g,1) | $(e,0) \to (d,1), (e,0) \to (g,1)$ |
| 4 | (f,0), (c,0) | $(d,1) \to (f,0), (d,1) \to (c,0)$ |
| 5 | (e,1), (f,1), (a,1) | $(f,0) \to (e,1), (c,0) \to (f,1), (c,0) \to (a,1)$ |
| 6 | (g,0) | $(e,1) \to (g,0)$ |

(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.)

*Proof.* We can use reduction to show that this game can be solved in $O(n^3)$ time using each player's unique capabilities.

Preprocess: To do this, we need to duplicate the board and its obstacles three times such that each player has their own board. Each of the $n^2$ spaces in the boards will be

a coordinate $(x, y)$ where x is the row and y is the column value. Then, using the techniques outlined in problems 3a and 3b, we could convert each of the players' boards into a graph where the coordinates are the vertices and the given player's possible moves from one coordinate to another are the edges. Thus, this step would take a total of $O(3n^2)$ time since there are 3 players and each one's board has $n^2$ vertices to transfer to a graph.

Oracle: Having reduced the problem to a Shortest Rotating Walk subroutine, for each player, we must identify their unique edges. Let's observe each player:

- Player 0 (moves vertically and horizontally): From any one vertex, the player can move $n-1$ spaces horizontally and $n-1$ spaces vertically (total: $2n-2$). Thus, from all $n^2$ vertices, this player can move $n^2(2n-2)$ different ways (and thus have this many edges in their graph).
- Player 1 (moves diagonally): From any one vertex, the player can move $n-1$ spaces in one diagonal direction and $n-1$ spaces in another diagonal direction (total: $2n-2$). Thus, from all $n^2$ vertices, this player can also move $n^2(2n-2)$ different ways (and thus have this many edges in their graph).
- Player 2 (moves 1 space vertically and 2 horizontally  vice versa): From any one vertex, the player can move at most 8 different ways. Thus, from all $n^2$ vertices, this player can move $8n^2$ different ways (and thus have this many edges in their graph).

Having gone through the enumeration of each player's edges, and now employing the technique used in previous problems (combining each rotation walk into one $G'$ graph) we can see that transferring these edges to $G'$ will take in the worst-case $O(n^3)$ time.

Postprocess: Adding the time it would take for us to transfer back the vertices and edges together, we observe that the entire process is dictated by the run-time of computing the edges. So the overall run-time of this entire algorithm would be $O(n^3)$ time.
Q.E.D.