CS1200: Intro. to Algorithms and their Limitations Anshu & Vadhan

Problem Set 4

Harvard SEAS - Fall 2024 *Due: Wed Oct. 9, 2024 (11:59pm)*

Your name: Ethan Veghte
Collaborators:
No. of late days used on previous psets: 5
No. of late days used after including this pset: 7

1. (Randomized Algorithms in Practice)

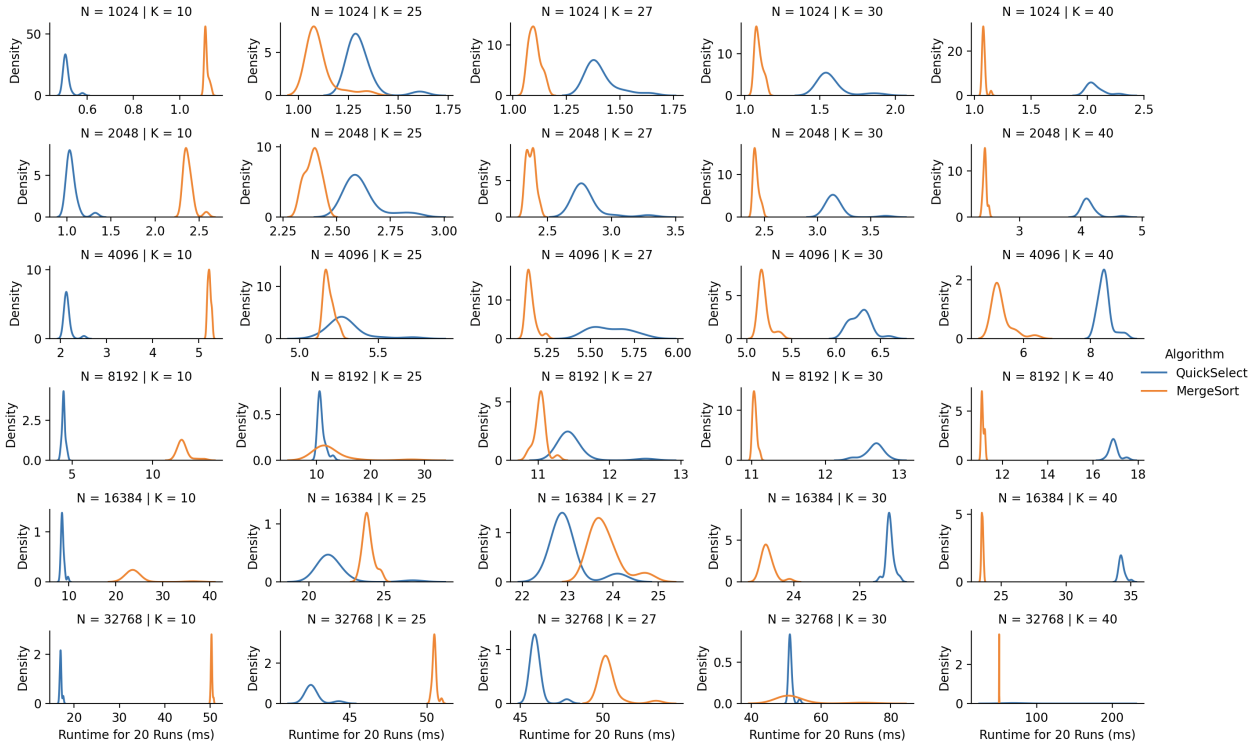
   (a) Implement Randomized QuickSelect, filling in the template we have given you in the Github repository.

   (b) In the repository, we have given you datasets x_n of key-value pairs of varying sizes to experiment with: dataset x_n is of size n. For each dataset x_n and any given number k, we will consider how to efficiently answer the k selection queries `select`$(x_n, 0)$, `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. For example, if $k = 4$, then we release the minimum, the 25th percentile, and the 75th percentile of the dataset. You will compare the following two approaches to answering the queries:

      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 `QuickSelect()` is randomized, and because of variance in the execution environment (e.g. other processes that 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 `QuickSelect()` to `MergeSortSelect()` for each given value of n (you can choose any reasonable statistical feature to propose $k^*(n)$, such as the peak runtime of the distribution or the mean runtime, etc). 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()`.)



This is a large amount of graphs to wrap your brain around, but as we can see, as k increases `MergeSortSelect()` *becomes increasingly more efficient compared to* `QuickSelect()`*. Just take the top row: when k is 1,* `QuickSelect()` *is far more efficient, but when k = 25, they are comparable. Then, when k = 40,* `MergeSortSelect()` *is much better. Another dimension to these graphs is what happens as n changes. We can see generally, that as N increases, the efficiency is maintained more in* `QuickSelect()` *than in* `MergeSortSelect()`*. This is most visible in the K =25 example, where,* `MergeSortSelect()` *is faster with lower N, but slower at higher N. I generally used peak runtime of the distribution (i.e. which runtime is most common), but also looked at variance of distribution as an important factor when making my choices below.*

Now lets look at which values of K we should switch from `QuickSelect()` *to* `MergeSortSelect()` *for differing values of N:*

N = 1024: At K = 10 `QuickSelect()` *is faster, but by K = 25,* `MergeSortSelect()` *is faster, so I would say probably around the K = 20 point.*

N = 2048: Very similar rationale as N = 1024, this case I would say K = 21.

N = 4096: Again, `MergeSortSelect()` *is slower at K = 10, but faster at K = 25. However, it is not that much faster, at that point. The density graph shows that it is more consistent though, making it more trustworthy to be efficient that* `QuickSelect()` *so I would say to use* `MergeSortSelect()` *at K = 25 for N = 4096.*

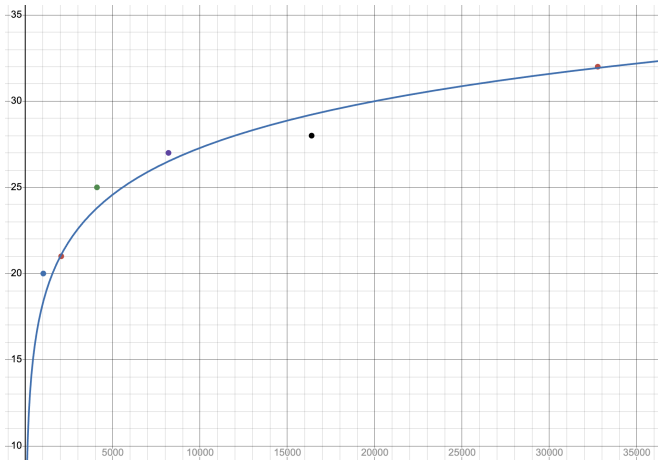
N = 8192: At K = 25, they are very comparable. In fact, for some reason `MergeSortSelect()` *is less densely distributed here, so I would still choose* `QuickSelect()`*. By K = 27, though,* `MergeSortSelect()` *is faster, so I would choose that as the break off point.*

N = 16384: At K = 27, `QuickSelect()` *is still consistently faster, but by K = 30 it is quite a bit slower. I will call K = 28 the turnover point.*

N = 32768: At K = 30 they are very comparable but with `QuickSelect()` *being much more consistent. By K = 40, the opposite is true, thus there is likely a point in the middle, say K = 32 where I would switch from* `QuickSelect()` *to* `MergeSortSelect()`.

(c) Extrapolate to come up with a simple functional form for $k^*(n)$, e.g. something like $k^*(n) = 3\sqrt{n} + 6$ or $k^*(n) = 10\log^2 n$. (Again there is not one right answer.) Briefly discuss how your extrapolation aligns with theoretical values. That is, what kind of functional form for $k^*(n)$ (in asymptotic notation) would be predicted by the asymptotic runtimes of `QuickSelect()` and `MergeSortSelect()` for answering k selection queries on a dataset of size n?

A simple functional form i found to represent $k^*(n)$ *was* $k^*(n) = 9\log(0.5n) - 6$. *I found this by plotting the intersection points of the runtimes of* `QuickSelect()` *and* `MergeSortSelect()` *and just experimenting on a function that seems like it accurately characterizes the points. This is one that I landed on.*



(d) (*optional) One way to improve `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. That is, in what way (if any) does `Median-of-3 QuickSelect()` offer benefits over `QuickSelect()`?

2. (Dictionaries and Hash Tables) Consider the following computational problem:

Input	**:** An array $(a_0, \ldots, a_{n-1})$ of natural numbers (each fitting in one word).
Output	**:** A duplicate element; that is, a number a such that there exist $i \neq j$ such that $a_i = a_j = a$.

Computational Problem DuplicateSearch

(a) Show that DuplicateSearch can be solved by a Las Vegas algorithm with expected runtime $O(n)$ using a dictionary data structure. (You should prove correctness and analyze runtime quoting the expected runtimes stated for the Dictionary data structure in Lecture 9, but you do not need to do a formal probability calculation using expectations.)

We learned that using a dictionary data structure combined with a Las Vegas algorithm we can have constant time operations. The method with which you would do Duplicate search would be to initialize the dictionary called D, then for each element, you search whether it already exists in D, if not you add it to D (O(1)), else you return that element as a duplicate. With this method you initialize, an empty dictionary, then search (O(1)), then possibly insert (O(1)). This process of searching then either returning or inserting, can happen in worst case for all elements of original array. That means the constant time operations happen across all elements making the runtime O(n). This shows that DuplicateSearch can be solved by a Las Vegas algorithm, due to randomization in a hash function for the dictionary implementation. This can be solved with expected runtime O(n) using a dictionary data structure. The algorithm is always correct, and its runtime is randomized with an expected linear time complexity.

(b) DuplicateSearch can be solved by a deterministic algorithm in runtime $O(n \log n)$. Briefly describe this algorithm in 2-3 sentences (you do not need to write a pseudocode and do not need to provide a proof of correctness).
The deterministic algorithm for DuplicateSearch would be to use binary search and iterate over every element in the array. That means for array A, you start at $A[0]$*, you are able to see whether it exists else where using Binary search on* $A[1:]$ *in* $O(\log n)$ *time. However, to find all the duplicates you must repeat this for all n elements of the list. This means you do* $\log n$ *operations on n elements* $\Rightarrow O(n \log n)$

3. (Choosing Algorithms and Data Structures) Suppose the US Census Bureau was going to develop a new database to keep track of the exact ages of the entire US population, and publish statistics on it. The data it has on each person is an exact birthdate `bday` (year, month, and date) and a unique identifier `id` (e.g. social security number — pretend that these are assigned at birth).

   For each of the three scenarios below,

   (a) Select the best algorithm or data structure for the Census Bureau to use from among the following:
   - sorting and storing the sorted dataset
   - storing in a binary search tree (balanced and possibly augmented)
   - storing in a hash table
   - running randomized QuickSelect.

   (b) Explain how you would use the algorithm or data structure (including any necessary augmentations) to solve the stated problem, e.g. what would you take as keys and values, what updates and queries (in case you use a dynamic data structure) would you issue, and how you would read off the results to obtain the desired statistics.

   (c) State what the runtime would be as a function of all of the relevant parameters: the size n of the US population being surveyed, the number u of updates issued at the specified time intervals, and/or the number s of statistics released at the specified time intervals. (These parameters are not all freely varying in the parts below, e.g. s may be a fixed constant or a function of n; state any such constraints in your answers.)

In each scenario, you should assume (unrealistically) that the described queries or statistics are the *only* way in which the data is going to be used, so there is no need to support anything else.

(a) (Reporting Age Rankings) Every ten years as part of the Decennial Census, the Bureau collects a fresh list of (`id`, `bday`) pairs from the entire US population. (It does not reuse data from the previous Decennial Census, so everyone is re-surveyed.) In order to incentivize participation, the Bureau promises to tell every respondent their age-ranking in the population after the survey is done (e.g. "you are the 796,421'th oldest person among those who responded to the Census").

For this situation, I see a few things. Most important, these operations are done every 10 years (i.e infrequently), but when it is done, a lot of information on the indices of certain id numbers must be pulled. I would choose to sort and store the sorted dataset. This is because if it is sorted based off of birthday (key), it would be ordered so that to get x in that 'you are the xth oldest person,' you simply retrieve the index + 1 of that person. Since you are doing this to every person, it is not a matter of lookup speed. Once the data is sorted and stored you would just iterate through every element in list, and retrieve the index. You do not need to lookup based on id or age, you just need to retrieve the index of everyone and distribute that information to them. It is a simple implementation in which once the data is gathered, a sorting algorithm like Merge Sort could sort the data in $O(n \log n)$ time. Once sorted, you are essentially done, with all significant computational problems. Thus even though it is expensive to sort the data, it only needs to be done once every ten years, and once finished, the structure becomes easy to traverse to get information for every person.

I have generally already covered runtime, but to reiterate: since the updates are every 10 years on every person, $u = n$, and it happens in $O(n \log n)$ time, as the sorting algorithm is the asymptotically most expensive element of the initialization. After, that you are forced to do a slow operation of iterating through every element, however, this is necessary no matter the data structure, so even though it happens in $O(n)$, nothing can be done to make it faster.

(b) (Daily Quartiles) After each day, the Bureau obtains a list of (`id`, `bday`) pairs to add to or remove from its database due to births, deaths, and immigration, and publishes an updated 25th, 50th, and 75th percentile of the population ages.

The primary factors that influence the data structure for the decision are that there are daily updates, and that percentile ranks need to be pulled easily. This is perfect for a balanced and augmented BST, because insertions and deletions are decently fast, and so are percentile pulls. Hash tables are difficult to do percentiles because they are unordered, and sorting and storing the sorted dataset is not ideal because the insertions and deletions are slower (and yet they must be done often in this scenario). We know that BSTs have $\log n$ *time for insertions and deletions, so can be satisfied with that speed. We also know that augmented and balanced BSTs can perform the percentile searches in* $\log n$ *time as well.*

The implementation, is to store birthdays as the key, and ids as the value. By making birthdays the key, the BST is ordered according to them, making it easy to gather the

relevant information for age percentiles. The ids are somewhat irrelevant in this situation because the only info needed is at a population level. However, it is likely important to keep for quality control purposes if it ever becomes necessary to check to ensure there are no repeated ids. Further, it is important that we properly augment and balance the tree, for the cases of percentile pulls. The augmentation is to have nodes hold data on the size of its subtree. In doing this, you know from origin node how big the entire tree is (call it T), and then if you were to traverse to left branch and see highest node on left branch and get subtree size (call it S), the percentile rank of origin node is $(S+1)/T$. *This is a much easier calculation than if it was not augmented and needing to count size of subtrees every time (O(n)). Also, important to note is that balancing the tree properly ensures that each side of tree is* $\log n$ *making operations faster.*
When investigating the runtimes, you know that each update (insertion or deletion) occurs in $\log n$ *time, as stated above. Thus if there are u, daily updates, the runtime of the updates are* $O(u * \log n)$. *The percentile rank publications is s statistics, and in this case just 3 as it sthe 25th, 50th and 75th percentiles, and each is calculated in* $O(\log n$ *time, so statistics are found in runtime* $O(s * \log n) = O(3 * \log n)$.

(c) (Age Lookups) For privacy reasons, the Bureau decides to not publish any statistics on the population ages, but just wants to maintain a database where the age of any member of the population can be looked up quickly, and the database can be quickly updated daily according to births, deaths, and immigration.

This is a perfect use case for storing in a hash table. The database is updated daily, which is easily done in a hash table, and the only look ups that need to be done to the database is at an individual level as opposed to percentiles or ranges. To implement this, I would use the id numbers of the individuals as the key, and the birthday as the value. It would be counter intuitive to assume that the person doing a search would have the birthday of a person they want to find the age of, so it is clearly the id that must be the key. Thus, I am going to assume that there is a way for people to retrieve the id numbers elsewhere for a specific person. This is somewhat unusual as SSNs for example are not public knowledge, but I am going to make this assumption, or else there would be no solution. Along these line, given a person has an id number in mind, they are able to query this database for the id number and be returned the age of that person. Given the searches are returning ages, a simple bit of arithmetic must be done to the birthday before returning it, as we stored the birthday as the value not the age. Furthermore, it is important that we store birthdays as opposed to ages, because if we stored ages, they would need to be updated everyday to accurately represent the person's age. Next, in order to update the database, it is just a series of insertions and deletions. Births are insertions, assuming an id has been assigned to the newborn, then deaths and immigration are deletions. Important implementation elements for updates are that there are checks to ensure there are no repeated ids, and that deletions occur on ids that exist.
The runtime of the insertions, and deletions that occur everyday are in constant time. Lets say the number of updates is u, so since each takes constant time, then the runtime of the updates on a given day are on average of O(u). Same can be said about the number of statistics, s, pulled on any given day. Since they each take constant time then runtime of the statistic pulls on any given day are expected to be of O(s). Of note, though, is that these are

expected runtimes. Because of the randomness of hash tables, there is a chance of having the worst case runtimes of $O(n)$ for these individual computation problems, however, this is a very very slim possibility.

4. (Reflection) Skim the course material from the beginning of the course through Lecture 9 (i.e. the scope of the upcoming class midterm). Identify one concept or skill that you would like to study or practice in greater depth, and discuss why. It can be because you feel that you haven't fully understood or internalized it, or because you found it interesting and are curious to learn more, or any other motivation you have.

   Note: As with the previous psets, you may include your answer in your PDF submission, but the answer should ultimately go into a separate Gradescope submission form.

   Quick note on grading: Good responses are usually about a paragraph, with something like 7 or 8 sentences. Most importantly, please make sure your answer is specific to this class and your experiences in it. If your answer could have been edited lightly to apply to another class at Harvard, points will be taken off.

   One concept that I hope to learn more about is hash tables. The interesting thing to me is that I see the incredibly broad use-case potential for them. It just seems like such a powerful tool to use, and am interested in knowing which underlying data structures I use on a day to day basis are hash tables. However, I still I struggle to understand how the randomness in selecting the hash function, ensures an expected runtime of $O(1 + n/m)$. It would seem to me that once the hash function is chosen, albeit randomly, the randomness of the structure becomes moot. The chosen hash function, h_l, has to be consistent to ensure that search brings you to the correct linked list thus it seems like randomness should no longer play a role in the runtimes. However, based on my understanding from lecture, it seems like it does. I would be excited to try implementing a more complex one than in CS50 should we be tasked with it, as I think that would be helpful for me to wrap my brain around it.
   After reading a little more on the concept, I want to write out what I learned for my own sake. I think where I tripped up was that in knowing you will implement a hash table as your data structure before receiving your data, you have randomness. Sometimes the data you receive can have patterns that would make collisions more likely should you pick and choose the hash function (thinking you know which would avoid the patterns), however, this does not always work. The randomness in choosing the function, thus guarantees an ***expected*** *uniform distribution across the buckets. This is importantly an expectation that randomness will reduce collisions, not a guarantee. Collisions are what slow it down, and thus to have a high degree of probability of reducing collisions in a also reduced memory size, compared to data structure like in Singleton Bucket Search (not sure name of data structure), is what makes hash tables so powerful.*

5. Once you're done with this problem set, please fill out this survey so that we can gather students' thoughts on the problem set, and the class in general. It's not required, but we really appreciate all responses!