CS1200: Intro. to Algorithms and their Limitations Prof. Salil Vadhan

Problem Set 4

Harvard SEAS - Fall 2025 *Due: Wed Oct. 8, 2025 (11:59pm)*

Please see the syllabus for the full collaboration and generative AI policy, as well as information on grading, late days, and revisions.

All sources of ideas, including (but not restricted to) any collaborators, AI tools, people outside of the course, websites, ARC tutors, and textbooks other than Hesterberg–Vadhan must be listed on your submitted homework along with a brief description of how they influenced your work. You need not cite core course resources, which are lectures, the Hesterberg–Vadhan textbook, sections, SREs, problem sets and solutions sets from earlier in the semester. If you use any concepts, terminology, or problem-solving approaches not covered in the course material by that point in the semester, you must describe the source of that idea. If you credit an AI tool for a particular idea, then you should also provide a primary source that corroborates it. Github Copilot and similar tools should be turned off when working on programming assignments.

If you did not have any collaborators or external resources, please write 'none.' Please remember to select pages when you submit on Gradescope. A problem set on the border between two letter grades cannot be rounded up if pages are not selected.

Your name:
Collaborators and External Resources:
No. of late days used on previous psets:
No. of late days used after including this pset:

1. (Randomized vs. Deterministic Hash Tables)

   (a) In the Github repository, we have given you a partial implementation of a `HashTable` data structure for the dynamic DICTIONARY abstract data type. When you initialize an object of the `HashTable` class, you specify the universe $[U]$ for keys, the size m of the hash table, a class of hash functions, and a boolean flag `optimize` to indicate whether to use the optimization described below. We have given you two classes of hash functions `DeterministicHash` and `RandomHash`; the first implements the common $h(x) = x \bmod m$ and the second implements $h_{a,b}(x) = ((ax+b) \mod p) \bmod m)$, where $p \geq U$ is a prime number and a and b are numbers in $\{0, 1, \ldots, p-1\}$ chosen uniformly at random at time at initialization.

   When the `optimize` flag is turned on, `HashTable` has our `search` and `delete` implementations always return or delete the key-value pair at the head of the linked list at the $h(K)$'th array entry. This is to save the time of searching the linked list. Unfortunately, as you will see in experiments below, this sometimes leads to erroneous results.

   Implement `HashTable` for the case when `optimize` is turned off, ensuring that it provides an always-correct DICTIONARY data structure.

   (b) We have also given you testing code (experiments.py) that carries out 10,000 operations on a `HashTable` data structure, maintaining a (fake) employee dataset consisting of pairs (`employeeid`, `salary`), where both `employeeid` are `salary` are given as integers in

$[10,000,000]$. The testing code calculates (a) the total time to carry out all operations, and (b) the number of times a search operation yields an incorrect result.

The testing code runs experiments with 100 trials for each combination of the following parameters:

- Whether we use `employeeid` or `salary` as the key.
- The size m of the hash table, with $m = [1, 10, 100, 1000]$.
- Whether or not `optimize` is turned on.
- Whether we use `DeterministicHash` or `RandomHash`.

Note that to run the experiments you simply need to run the experiments.py file and not make any changes to it.

Using the results of these experiments, discuss the following:

i. Were any combinations of parameters particularly slow in an answering individual queries? Why do you think that was the case?

ii. Were any combinations of parameters prone to errors, and how frequently? Why do you think that was the case? For each combination of `optimize` and choice of the family of hash functions, classify the resulting algorithm as being a correct deterministic algorithm, a correct Las Vegas algorithm, a correct Monte Carlo algorithm, or an incorrect algorithm.

iii. What are benefits or costs you see to using randomization for hash tables?

iv. Are the runtimes you observed consistent with the $O(1+n/m)$ time bound for hash tables discussed in class?

2. (Weighted Medians) A local jazz club wants to determine the median age of the people that attend its shows. It has a dataset consisting of each customer's age and the number of club visits they have made during the past 5 years. When calculating the median, they want to put more weight on the customers that visit the club more often (since they have a bigger effect on the ages present in the club). This leads to the following computational problem:

Input: An array A of key-value-weight triples $((K_0, V_0, W_0), \ldots, (K_{n-1}, V_{n-1}, W_{n-1}))$, where $K_i, W_i \in \mathbb{N}$, and a threshold $t \in \mathbb{N}$
Output: A key-value-weight triple (K_i, V_i, W_i) such that $\sum_{j:K_j<K_i} W_j \leq t$ and $\sum_{j:K_j>K_i} W_j \leq W - 1 - t$, where $W = \sum_{j=0}^{n-1} \mathrm{W}_j$

Computational Problem Weighted Selection

For motivating application, consider the keys to be the customer ages, the weights to be the number of times they visited the club, and $t = W/2$.

Write out (in pseudocode) a modification of `QuickSelect` that solves the Weighted Selection problem. Note that unlike Selection, where there is always a solution when the rank $i \in [n]$, there may be cases where Weighted Selection has no solution and your algorithm should indicate so with $\perp$. Prove the correctness of your algorithm, then informally justify why the worse-case expected runtime of $O(n)$ derived from lecture is still valid for this modification.

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 (optionally, four) scenarios below,

   [label=()]Select the best algorithm or data structure for the Census Bureau to use from among the following:

   (a)
   - 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. If you find that only `id` or `bday` is relevant or necessary for some part, you can feel free to ignore the other component in your data structure.

Example (Decennial Quartiles): Every ten years as part of the Decennial Census, the Bureau obtains a fresh list of (`id`, `bday`) pairs for the entire US population and wishes to release the 25th, 50th, and 75th percentiles of the population ages (to the day).

Solution:

[label=()]**Recommendation:** I would recommend that the Census Bureau use `QuickSelect`. I guess that a large enough of a fraction of the US population changes every 10 years that processing the updates individually would be more expensive than doing batch computations. **How I would use my data structure/ algorithm:** They should use the `bday`'s as keys; the `id`'s are not needed because we only want to release age percentiles. Each time they should run `QuickSelect` with ranks $i = n/4, n/2, 3n/4$, where n is the size of the population at the time. **Runtime:** We are calculating $s = 3$ statistics every 10 years, and each execution of `QuickSelect` takes expected time $O(n)$, for a total runtime of $s \cdot O(n) = O(n)$.

(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").

(b) (Daily Quartiles) After each day, the Bureau obtains a list of data (given as 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.

(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.

(d) (Daily Quartiles, optional[1]) The Bureau receives a daily list of updates and publish the 25th, 50th, and 75th percentile of population ages, similar to (b). This time, however, when an update deletes someone's data (i.e. death, emigration), the Bureau only receives their `id`. Incoming data (i.e. birth, immigration) is still in the format (`id`, `bday`).

4. (Reflection) Skim the course material from the beginning of the course through Lecture 10 (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.

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!

1 Old/Backup Problems

6. (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 $\texttt{select}(x_n, 0)$, $\texttt{select}(x_n, [n/k])$, $\texttt{select}(x_n, [2n/k])$, ..., $\texttt{select}(x_n, [(k-1)n/k])$ 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.

[1] This problem won't make a difference between N, L, R-, and R grades.

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()`.)

(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?

(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()`?

7. (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) Give a brute-force deterministic algorithm to solve DuplicateSearch in $O(n^2)$. Describe this algorithm in 2-3 sentences (you do not need to write pseudocode or provide a proof of correctness).

(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).

(c) 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.)