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

Lecture 10: Randomized Data Structures, Synthesis

Harvard SEAS - Fall 2025 *2025-10-02*

1 Announcements

- Salil's today 12:30pm-1:15pm, SEC 3.327.
- SRE 3 moved to Thursday 10/9.
- Interrupt me when I start writing too small!
- Participate more in lecture! There are no bad questions or answers. You can also write them in Ed and our staff will answer or read aloud. Talk to your neighbor when I give the class a question to think about.

Recommended Reading:

- Hesterberg–Vadhan Ch. 10
- CLRS 11.0-11.4
- Roughgarden II,12.0-12.4

2 Loose Ends: Analysis of QuickSelect

Theorem .1. *There is a randomized algorithm, called* `QuickSelect`*, that always solves* SELECTION*correctly and has (worst-case) expected running time* $O(n)$*.*

Proof Sketch. 1. The algorithm:

```
QuickSelect($A, i$):
Input        : An array $A = ((K_0, V_0), \ldots, (K_{n-1}, V_{n-1}))$, where each
               $K_j \in \mathbb{N}$, and $i \in [n]$
Output       : A key-value pair $(K_j, V_j)$ such that $K_j$ is an
               $(i+1)^{\text{st}}$-smallest key.
0 if $n == 1$ then return $(K_0, V_0)$;
1 $p = \texttt{random}(n)$;
2 pivot $= K_p$;
3 Let $A_{\text{smaller}}$ = an array containing the elements of $A$ with keys < pivot;
4 Let $A_{\text{larger}}$ = an array containing the elements of $A$ with keys > pivot;
5 Let $A_{\text{equal}}$ = an array containing the elements of $A$ with keys = pivot;
6 Let $n_{\text{smaller}}, n_{\text{larger}}, n_{\text{equal}}$ be the lengths of $A_{\text{smaller}}$, $A_{\text{larger}}$, and $A_{\text{equal}}$ (so
    $n_{\text{smaller}} + n_{\text{larger}} + n_{\text{equal}} = n$);
7 if $i < n_{\text{smaller}}$ then return QuickSelect($A_{\text{smaller}}, i$);
8 else if $i \geq n_{\text{smaller}} + n_{\text{equal}}$ then return
    QuickSelect($A_{\text{larger}}, i - n_{\text{smaller}} - n_{\text{equal}}$);
9 return $A_{\text{equal}}[0]$
```

Algorithm .1: `QuickSelect()`

Example .2. Let $A = [2, 3, 5, 6, 3, 5, 4]$ with $n = 7$ and $i = 3$ (here we only keep track of the keys). Note that the 4th smallest key is 4.

We first randomly pick the pivot $p = 2$

2. **Proof of correctness sketch:**

3. **Expected runtime:**

   Given an array of size n, the size of the subarray that we recurse on is bounded by $\max\{n_{\text{smaller}}, n_{\text{larger}}\}$. On average over the choice of our pivot, the subarrays will be fairly balanced in size, and indeed it can be shown that

   $$\mathrm{E}\left[\max\{n_{\text{smaller}}, n_{\text{larger}}\}\right] \leq$$

   Thus, intuitively, the expected runtime of QuickSelect should satisfy the recurrence:

   $$T(n) \leq$$

   for $n > 1$. Then by unrolling we get:

   $$T(n) \leq$$

□

3 Loose End: The Power of Randomized Algorithms

A fundamental question is whether allowing randomization (in either the Las Vegas or Monte Carlo ways) actually adds power: are there problems with faster randomized solutions than the fastest deterministic ones? Here are some examples of problems that reflect a potential gap between randomized and deterministic algorithms:

- Selection: Theorem .1 gave a simple $O(n)$ time Las-Vegas algorithm. There *is* a deterministic algorithm with runtime $O(n)$, which uses a more complicated strategy to choose a pivot (and is a constant factor slower).
- Primality Testing: Given a bignum integer of size n words of memory, check if it is prime. There is an $O(n^3)$-time Monte Carlo algorithm, $O(n^4)$-time Las Vegas algorithm, and a $O(n^6)$-time deterministic algorithm (proven in the paper "Primes is in P").
- Identity Testing: Given some arithmetic expression with bignum exponents (like $12875^{8091761235} \cdot (15676 + 91247^{259}) - 967012^{8107069871}$), check if it is equal to zero. This has an $O(n^2)$-time Monte Carlo algorithm, and the best currently-known deterministic algorithm runs in $2^{O(n)}$!

Nevertheless, the prevailing conjecture is:

You can learn more about this conjecture in courses like CS1210, CS2210, and CS2253.

4 Randomized Data Structures: Dynamic Dictionaries

We can define randomized data structures to be randomized by allowing the algorithms Preprocess, EvalQ, and EvalU to be randomized algorithms. Like randomized algorithms, randomized data structures can be either Las Vegas (never make an error, but have random runtimes) or Monte Carlo (fixed runtimes, but can err with small probability).

A canonical data structure problem for which randomization is useful is the *dictionary* problem:

Updates: `insert` or `delete` a key-value pair (K, V) with $K \in [U]$ into a multiset

Queries: `search`(K) - return a matching key-value pair (K, V) from the multiset (if one exists)

Abstract Data Type (Dynamic) DICTIONARY

Note that in the DICTIONARY problem we fix a universe size U for the keys upfront; this should be in the initial *input*, given before any key-value pairs are inserted.

Q: What data structure have we seen that already solves the dynamic DICTIONARY problem?

A: We will be more ambitious and aim for $O(1)$ time for all of our operations, if

we allow randomization:

Theorem .3. *Given a positive integer m as a parameter, the* DICTIONARY *dynamic abstract data type has a Las Vegas randomized data structure,* `HashTable`*, in which* `insert`*,* `delete`*, and* `search` *take expected time ____________, where n is the number of elements in the multiset currently being stored. Preprocessing involves initializing an empty block of memory of size $O(m)$ and $O(1)$ additional runtime.*

Whenever $m = \Theta(n)$, the expected runtime of updates and queries is $O(1)$. If n becomes larger, one has to update the data structure by choosing a larger value of m. This change from $O(\log n)$ time per query to $O(1)$ time per query makes a major difference in real-world high-performance computer systems.

As a warm-up, let's see a simpler, deterministic data structure for the case that the universe size U is not too large.

Proposition .4. *The* DICTIONARY *dynamic abstract data type has a data structure in which* `insert`*,* `delete`*, and* `search` *take $O(1)$ time and preprocessing involves initializing an empty block of memory of size $O(U)$ and $O(1)$ additional time.*

Proof. The data structure is as follows

- Preprocess:

- Insert:

- Delete:

- Search:

Runtime:

□

Q: Where have we seen this data structure before?

A:

Reducing preprocessing time: If U is very large, then the preprocessing cost of the above algorithm is prohibitive. We might try to reduce the preprocessing time as follows: rather than allocating an array of size U, allocate one of a smaller size $m = O(n)$ (say, $m = 4n$), and send the key (K, v) to a linked list at, not array position K, but array position $h(K) = K \;\%\; m$.

Q: What does wrong with this idea and how can we fix it?

A:

Q: What new problem does this fix introduce?

A:

Here randomization comes to the rescue! A randomized hash function can guarantee that the expected lengths of the linked lists and our runtime will be $O(1)$ even on worst-case datasets. An example of a randomized hash function that works well is the following: pick a prime number p with $U \leq p < 2U$. Then, at initialization, we choose uniformly random $a \in \{1, \ldots, p-1\}$ and $b \in \{0, \ldots, p-1\}$, and obtain the hash function

$$h_{a,b}(K) = \underline{\hspace{4cm}} \tag{1}$$

The properties we need from this construction (or other constructions of random hash families) are abstracted as follows:

Lemma .5 (random hash functions)**.** *For all natural numbers U, m with $m \leq U$, there exists a set $\mathcal{R}$ and, for every $\ell \in \mathcal{R}$, a function $h_\ell : [U] \to [m]$, with the following properties:*

- Compact description:

- Efficient evaluation:

- Low collision probability:

Remarks:

- For the construction from Equation (1), we take $\mathcal{R} = [p] \times [p]$, so $|\mathcal{R}| = p^2 \leq 4U^2$ and it satisfies efficient evaluation by inspection (it just involved $O(1)$ arithmetic operations on numbers that fit in $O(1)$ words). We omit the proof that it has low collision probability, which requires a bit of number theory.
- If U does not fit in $O(1)$ words (e.g. we are hashing URLs or long phrases), then the efficient evaluation property should be relaxed to time $u^{O(1)}$, where u is the number of words for each element of U.

Proof of Theorem .3. The algorithm is as follows
.

- Preprocess:

- Insert:

- Delete:

- Search:

Correctness: The algorithm correctly maintains a multiset and answers search queries, by construction. Hence it is a Las Vegas data structure.

Expected Runtime: The preprocessing step initializes an empty array of size $O(m)$ and chooses a random number ℓ in time $O(1)$.

The insert operation evaluates $h_\ell(K)$ and add (K, V) at the start of the linked list, each of which takes time $O(1)$.

The runtime of the delete and search operations is:

$\square$

Load of a hash table. We call $\alpha = n/m$ the *load* of the just-defined data structure, so the expected runtimes of the delete and search operations are both $O(1+\alpha)$. Notice that that expected time is $O(1)$, the best we could hope for, as long as $m = \Omega(n)$. To achieve this optimal time efficiency, we need to tailor m to the size n of the dataset, which we may not know in advance, and which changes as items are inserted and deleted. This issue can be solved by rebuilding the whole data structure whenever the hash table gets too full. This can be done in such a way that the average cost per key-value pair (referred to as the *amortized cost*) is $O(1)$.

5 Storing and Searching Data Synthesis

Q: We have seen several approaches to storing and searching in large datasets (of key-value pairs). For each of these approaches, describe a feature or combination of features it has that none of the other approaches provide.

1. Sort the dataset and store the sorted array

2. Store in a binary search tree (balanced and appropriately augmented)

3. Run Randomized QuickSelect

4. Store in a hash table

Maxwell's Flowchart. In general, to determine which algorithm or data structure to use for storing or searching data, the following flowchart can be a useful guide: