CS120: Intro. to Algorithms and their Limitations Hesterberg & Vadhan

Lecture 8: Randomized Algorithms and Data Structures

Harvard SEAS - Fall 2022 *2022-09-27*

1 Announcements

- Pset 1 grades returned
- Pset 3 due tomorrow.
- Add/drop 10-03
- Midterm in class 10-04

2 Randomized Algorithms

Recommended Reading:

- CLRS Sec 9.0–9.2, 11.0–11.4
- Roughgarden I Sec. 6.0–6.2
- Roughgarden II 12.0–12.4
- Lewis-Zax Chs. 26-29

2.1 Definitions

Recall that one criterion making the RAM and Word-RAM models we defined in the previous lecture are a solid foundational model of computation was expressivity; the ability to do everything we consider reasonable for an algorithm to do. However, as we've defined them, every run of a RAM or Word-RAM program on the same input produces the same output, which is not a characteristic of all Python programs:

One very useful ingredient to add to algorithms is *randomization*, where we allow the algorithm to "toss coins" or generate random numbers, and act differently depending on the results of those random coins. We can model randomization by adding to the RAM or Word-RAM Model a new `random` command, used as follows:

$$\texttt{var}_1 = \texttt{random}(\texttt{var}_2),$$

which assigns $\texttt{var}_1$ a uniformly element of the set $[\texttt{var}_2]$.

We assume that all calls to `random()` generate independent random numbers. (In reality, implementations of randomized algorithms use *pseudorandom number generators*. Courses such as CS 121, 127, 221, and 225 cover the theory of pseudorandom generators,[1] and how we can be sure

[1] CS 225 is not likely to be offered in the next couple of years, but you can learn the material from Salil's textbook *Pseudorandomness*.

that our randomized algorithms will work well even when using them instead of truly uniform and independent random numbers.)

There are two different flavors of randomized algorithms:

- *Las Vegas Algorithms:* these are algorithms that always output a correct answer, but their running time depends on their random choices. Typically, we try to bound their *expected* running time. That is, we say the *(worst-case) expected running time* of A is
$$T(n) = \max_{x:\text{size}(x)\leq n} \mathrm{E}[\mathrm{Time}_A(x)],$$
where $\mathrm{Time}_A(x)$ is the random variable denoting the runtime of A on x, and $\mathrm{E}[\cdot]$ denotes the expectation of a random variable:
$$\mathrm{E}[Z] = \sum_{z\in\mathbb{N}} z\cdot \Pr[Z=z].$$
- *Monte Carlo Algorithms:* these are algorithms that always run within a desired time bound $T(n)$, but may err with some small probability (if they are unlucky in their random choices), i.e. we say that A *solves* computational problem $\Pi = (\mathcal{I}, f)$ *with error probability* p if
$$\text{for every } x \in \mathcal{I},\ \Pr[A(x) \in f(x)] \geq 1-p$$
Think of the error probability as a small constant, like $p = .01$. Typically this constant can be reduced to an astronomically small value (e.g. $p = 2^{-50}$ by running the algorithm several times independently).

We stress that in both cases, the probability space is the sequence of random draws made by `random()`. We still do a *worst-case analysis* over inputs: we want to bound the expected running time of a Las Vegas algorithm on *all* inputs, and the error probability of a Monte Carlo algorithm on *all* inputs. A Las Vegas algorithm may run arbitrarily long and a Monte Carlo program may give an incorrect answer with sufficiently unlucky randomness.

Q: Which is preferable (Las Vegas or Monte Carlo)?

A: Las Vegas. It turns out that every Las Vegas algorithm can be converted into a Monte Carlo one with comparable runtime, but the converse is not known. However, it can be easier to come up with Monte Carlo algorithms. Testing whether a large (bignum) integer is prime is an example where we have Monte Carlo algorithms that are faster than any known Las Vegas algorithms.

2.2 QuickSelect

We will see an efficient Las Vegas algorithm for the following problem:

Input	**:** An array A of key-value pairs $((K_0, V_0), \ldots, (K_{n-1}, V_{n-1}))$, where each key $K_j \in \mathbb{N}$, and a *rank* $i \in [n]$
Output	**:** A key-value pair (K_j, V_j) such that K_j is an i'th smallest key. That is, there are at most i values of k such that $K_k < K_j$ and there are at most $n-i-1$ values of k such that $K_k > K_j$.

Computational Problem Selection

In particular, when $i = (n-1)/2$, we need to find the *median* key in the dataset.

Motivating Problem: These days, the algorithmic statistics and machine learning community has a lot of interest in medians (and high-dimensional analogues of medians) because of their *robustness*: medians are much less sensitive to outliers than means. We may want robustness to outliers because real-world data can be noisy (or adversarially corrupted), our statistical models may be misspecified (e.g. a Gaussian model may mostly but not perfectly fit), and/or for privacy (we don't want the statistics to reveal much about any one individual's data — take CS126, CS208, or CS226 if you are curious about this topic).

We can solve Selection in time $O(n \log n)$. How? Sort (time $O(n \log n)$) and return the i'th element of the sorted array (time $O(1)$).

But with randomization, we can obtain a simpler and faster algorithm.

Theorem 2.1. *There is a randomized algorithm 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 $i \in \mathbb{N}$
Output  : A key-value pair $(K_j, V_j)$ such that $K_j$ is an $i$'th smallest key.
if $n \leq 1$ then return $(K_0, V_0)$;
else
    $p$ = random($n$);
    $P = K_p$ ;                                  /* $P$ is called the pivot */
    Let $A_<$ = an array containing the elements of $A$ with keys $< P$;
    Let $A_>$ = an array containing the elements of $A$ with keys $> P$;
    Let $A_=$ = an array containing the elements of $A$ with keys $= P$;
    Let $n_<, n_>, n_=$ be the lengths of $A_<$, $A_>$, and $A_=$ (so $n_< + n_> + n_= = n$);
    if $i < n_<$ then return QuickSelect($A_<, i$);
    else if $i \geq n_< + n_=$ then return QuickSelect($A_>, i - n_< - n_=$) ;
    else return $A_=[0]$;
```

Algorithm 1: QuickSelect

Example: Let $A = [2, 3, 5, 6, 3, 5, 4]$ and $i = 3$ (here we only keep track of the keys). Here $n = 7$.

Our first random pivot is $p = 2$ so $K_p = 5$. We then divide the list into $A_< = [2, 3, 3, 4]$, $A_= = [5, 5]$ and $A_> = [6]$. Since $i < |A_<|$, we recurse on QuickSelect($A_<$,i).

2. Proof of correctness:
Essentially, no matter what we select as the pivot we correctly recurse on the subproblem, so a proof by induction on $n = |A|$ works to show correctness.

3. Expected runtime: we are only going to sketch the proof of this analysis, since we will not be asking you to do sophisticated probability proofs during the course. For CS120, our goal is for you to understand the concept of randomized algorithms and why they are useful. You can develop more skills in rigorously analyzing randomized algorithms in subsequent courses like CS121, CS124, and other CS22x courses. However, a full proof is enclosed in Section 2.4 of the detailed notes for optional reading in case you are interested.

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

$$\mathrm{E}\left[\max\{n_<, n_>\}\right] \leq \frac{3n}{4}.$$

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

$$T(n) \leq T(3n/4) + cn,$$

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

$$T(n) \leq cn + c \cdot \left(\frac{3}{4}\right) \cdot n + c \cdot \left(\frac{3}{4}\right)^2 \cdot n + \cdots = 4cn.$$

□

2.3 The Power of Randomized Algorithms

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

- Selection: 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 has a larger constant in practice).
- Primality Testing: Given an integer of size n (where n=number of words), check if it is prime. There is an $O(n^3)$ time Monte Carlo algorithm, $O(n^4)$ Las Vegas algo and $O(n^6)$ determinstic algorithm (proven in the paper "Primes is in P").
- Identity Testing: Given some algebraic expression, check if it is equal to zero. This has an $O(n^2)$ time Monte Carlo algorithm, and the best known deterministic algorithm runs in $2^{O(n)}$!

Nevertheless, the prevailing conjecture (based on the theory of pseudorandom number generators) is: **randomness is not necessary for polynomial-time computation.** More precisely, we believe a $T(n)$ time Monte-Carlo algorithm *implies* a deterministic algorithm in time $O(n \cdot T(n))$, so we can convert any randomized algorithm into a deterministic one. You can learn more about this conjecture in courses like CS121, CS221, and CS225.

2.4 Rigorous runtime analysis for QuickSelect

This section is optional, only in the detailed notes, only for reading out of curiousity!

Proof. Let $T(n)$ be the worst-case expected runtime of `QuickSelect()` on arrays of length n. Fix an array A of length $n \geq 1$. The proof combines the following three claims:

$$\mathrm{E}\left[\mathrm{Time}_{\mathrm{QuickSelect}}(A)\right] = \mathrm{E}_p\left[\mathrm{E}\left[\mathrm{Time}_{\mathrm{QuickSelect}}(A)|p\right]\right], \tag{1}$$

where $|p$ denotes conditioning on a fixed value of p.

$$\mathrm{E}_p\left[\mathrm{Time}_{\mathrm{QuickSelect}}(A)|p\right] \le cn + \max\{T(n_<), T(n_>)\} = cn + T(\max\{n_<, n_>\}). \tag{2}$$

$$\mathrm{E}_p[\max\{n_<, n_>\}] \le 3n/4, \tag{3}$$

Equation (1) is a general probability fact known as the Law of Iterated Expectations. Inequality (2) follows from inspection of the algorithm.

To prove Inequality (3), we observe that the lengths $n_<, n_>, n_=$ of the partitioned array depend only on $P = K_p$, and that the distribution of P remains the same regardless of the ordering of the elements of A. In particular, if we consider a *sorted* version $A' = ((K'_0, V'_0), \ldots, (K'_{n-1}, V'_{n-1}))$ of A and choose an index p' uniformly at random from $[n]$, then $P' = K'_{p'}$ has exactly the same distribution as P, and the lengths $n'_<, n'_>, n'_=$ of the corresponding partition of the sorted array has the same distribution as $n_<, n_>, n_=$.

With this choice, we observe that $n'_< \le p'$ and $n'_> \le n - p' - 1$, so we have the following (assuming n is odd for simplicity):

$$\begin{aligned}
\mathrm{E}_p[\max\{n_<, n_>\}] &= \mathrm{E}_{p'}[\max\{n'_<, n'_>\}] \\
&< \mathrm{E}_{p'}[\max\{p', n - p' - 1\}] \\
&= \frac{1}{n}\sum_{p'=0}^{n-1} \max\{p', n - p' - 1\} \\
&\le \frac{2}{n}\sum_{q=(n-1)/2}^{n-1} q \\
&= \frac{2}{n} \cdot \frac{n-1}{2} \cdot \frac{3(n-1)}{4} \\
&\le \frac{3n}{4}.
\end{aligned}$$

(The case of even n is similar.)

Equations/Inequalities (1), (2), and (3) suggest a recurrence like:

$$T(n) \le cn + T(3n/4),$$

which would imply that

$$T(n) \le cn + c \cdot \left(\frac{3}{4}\right) \cdot n + c \cdot \left(\frac{3}{4}\right)^2 \cdot n + \cdots = 4cn.$$

However, it is not true in general that for an arbitrary function f and random variable X that

$$\mathrm{E}[f(X)] = f(\mathrm{E}[X]) \tag{4}$$

(We are interested in $f = T$ and $X = \max\{n_<, n_>\}$.) However, this Equation (4) is true when the function f is linear, which is the bound we are trying to prove on T. So we can prove our desired bound that $T(n) \le 4cn$ for all natural numbers $n \ge 1$ by induction on n.

For the base case, we have $T(1) \leq 4c$ by inspection (for a large enough choice of the constant c. For the induction step, assume that $T(k) \leq 4ck$ for all natural numbers $1 \leq k \leq n-1$, and let's prove that $T(n) \leq 4cn$ for $n \geq 2$. That is, we need to show that for every array A of length n, the expected runtime of QuickSelect on A is at most $4cn$. We do this as follows:

$$\begin{aligned}
\mathrm{E}\left[\mathrm{Time}_{\mathrm{QuickSelect}}(A)\right] &= \mathrm{E}_p\left[\mathrm{E}\left[\mathrm{Time}_{\mathrm{QuickSelect}}(A)|p\right]\right] && \text{(Equation (1))}\\
&\leq \mathrm{E}_p[cn + T(\max\{n_<, n_>\})] && \text{(Inequality (2))}\\
&\leq \mathrm{E}_p[cn + 4c \cdot \max\{n_<, n_>\})] && \text{(Induction Hypothesis, since } n_<, n_> < n)\\
&\leq cn + 4c \cdot \mathrm{E}_p[\max\{n_<, n_>\}] && \text{(Linearity of Expectations)}\\
&= cn + 4c \cdot (3n/4) && \text{(Inequality (3)}\\
&= 4cn,
\end{aligned}$$

as desired.

□

3 Randomized Data Structures

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

A canonical data structure problem where randomization is useful is the *dictionary* problem. These are data structures for storing sets of key-value pairs (like we've been studying) but where we are *not* interested in the ordering of the keys (so min/max/next-smaller/selection aren't relevant).

Updates :	Insert or delete a key-value pair (K, V) with $K \in \mathbb{N}$ into the multiset
Queries :	Given a key K, return a matching key-value pair (K, V) from the multiset (if one exists)

Data-Structure Problem(Dynamic) Dictionaries

Of course the Dynamic Dictionary Problem is easier than the Predecessor Problem we have already studied, so we can use Balanced BSTs to perform all operations in time $O(\log n)$. So our goal here will be to do even better — get time $O(1)$.

Let's assume our keys come from a finite universe U. Then we can get $O(1)$ time as follows:

A deterministic data structure:

- Preprocess(U): Initialize an array A of size U.
- Insert(K, V): Place (K, V) into a linked list at slot $A[K]$.
- Delete(K): Remove the head of the linked list at slot $A[K]$.
- Search(K): Return the head of the linked list at $A[K]$.

A problem with this approach: U can be very large. If we consider keys of 64 bit words (common in practice), we would need an array of size 2^{64}, which is completely infeasible. Also, although this is $O(1)$ time per query, Preprocess(U) takes time $\Theta(U)$.

Attempted fix 1: Choose an integer m much smaller than U, make A of size m, and put (K, V) at spot $A[K\%m]$.

A problem with this approach: some sequences of operations will *always* fail, violating the requirements for a randomized algorithm. For instance, if $m = 128$, then after the operations Insert($(0, K)$) and Insert($(128, K)$), the operation Search(128) will always fail.

Attempted fix 2: Choose m as above, but choose some other function $h : [U] \to [m]$ as a replacement for %, and put a key K at $A[h(K)]$

Unfortunately, this approach has the same problem as the previous attempted fix.

Attempted fix 3: Choose a *random* function $h : [U] \to [m]$. Then the error probability is always small.

However, in this case, we again need a large amount of memory (whose size depends on U).

An actual fix: One more tweak makes this work: choose a "random hash function" $h : [U] \to [m]$ instead of a random function $h : [U] \to [m]$. For the purposes of CS 120, a *random hash function* is the same as a random function, except that:

1. Generating a random hash function h takes time $O(1)$.

2. Storing h takes space $O(1)$.

3. For all $x \in U$, evaluating $h(x)$ takes time $O(1)$.

Constructing and analyzing random hash functions is outside the scope of CS 120, but there is some optional reading on it in Section 3.1 below in case you are curious.

A Monte Carlo data structure:

- Preprocess(U, m): Initialize an array A of size m. In addition, choose a random hash function $h : [U] \to [m]$ from the universe $[U]$ to $[m]$.

- Insert(K, V): Place (K, V) into the linked list at slot $A[h(K)]$.

- Delete(K): Remove an element from the linked list at slot $A[h(K)]$.

- Search(K): Return the head of the linked list at $A[h(K)]$.

Unfortunately, this is a Monte Carlo data structure - if there are two distinct keys K_1, K_2 with $h(K_1) = h(K_2)$, on the query Search(K_1) we could return an key-value pair (K_2, V). To bound this error probability, consider a query Search(K). In order for us to return the wrong value, we must have inserted a distinct key that hashes to the same value as $h(K)$. If we've inserted items with keys $K_0, \dots, K_{n-1}$ that are different than K, we have:

$$\begin{aligned}
\Pr[\text{Search}(K) \text{ returns incorrect value}] &\le \Pr[h(K) \in \{h(K_0), \dots, h(K_{n-1})\}] \\
&\le \sum_{i=0}^{n-1} \Pr[h(K) = h(K_i)] \\
&= n \cdot \frac{1}{m}
\end{aligned}$$

where in the final line we used that h was a random mapping from $[U]$ to $[m]$.

A Las Vegas data structure ("Hash Table"):
The same data structure as above, except the linked list at every array index stores (K, V) pairs. Then when we query Search(K), go to the linked list $A[h(K)]$ and return the first element of the list that has the correct key K (and if none do, return $\perp$).

Here, we bound the *expected runtime* via the same analysis as before, because if no elements collide (the event we bound above) the additional linked list checking will only be an $O(1)$ slowdown. Quantitatively, we can show an expected runtime of $O(1 + n/m)$. We call $\alpha = n/m$ the *load* of the table, so the runtime is $O(1 + \alpha)$. Notice that here we get $O(1)$ expected time even if $n > m$, provided that $m = \Omega(n)$.

To maintain both time and space efficiency, we need to tailor the size m of the table to the size n of the dataset, which we may not know advance. This can be solved by dynamically resizing the dataset as the hash table gets too full. For example, when we reach load $\alpha = 2/3$, we can double the table size to bring us back to $\alpha = 1/3$.

3.1 Hash functions

This section is optional reading on how hash functions are constructed, in case you are interested. More aspects of this problem are discussed in the CLRS and Roughgarden texsts, and courses like CS 124, CS 127, CS 222, CS 223, and CS 225.

We want a family $\mathcal{H}$ of hash functions h, smaller than the family of all random functions, that allows us to (a) store h compactly, (b) evaluate h efficiently, and (c) still prove that the worst-case expected time for operations on the hash table is $O(1 + \alpha)$. An example: pick a prime number $p > U$. Then for $a \in \{1, \dots, p-1\}$ and $b \in \{0, \dots, p-1\}$ we define the hash function

$$h_{a,b}(K) = ((aK + b) \mod p) \mod m.$$

This takes 2 words to store, can be evaluated in $O(1)$ time, and maintains the same pairwise collision property: for every $K \neq K' \in [U]$, we have

$$\Pr_{a,b}[h_{a,b}(K) = h_{a,b}(K')] \le \frac{1}{m} \tag{5}$$

(For a proof, see CLRS. This requires a little bit of number theory and is beyond the scope of this course.) A hash family satisfying (5) is known as an *universal hash family*, and this property suffices to prove our expected runtime bounds of $O(1 + \alpha)$.

We could also use a "cryptographic" hash function like SHA-3, which involves no randomness but it conjectured to be "hard to distinguish" from a truly random function. (Formalizing this conjecture is covered in CS 127.) This has the advantages that the hash function is deterministic and that we do not need to fix a universe size U. On the other hand, the expected runtime bound is then based on an unproven conjecture about the hash function, and also these hash functions, while quite fast, are not quite computable in $O(1)$ time. By combining them with a little bit of randomization, they can also be made somewhat resilient against adversarial data, where an adversary tries to learn something about the hash function by interacting with the data structure and uses that knowledge to construct data that makes the data structure slow.