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

Lecture 8: Randomized Algorithms and Data Structures

Harvard SEAS - Fall 2023 *2023-09-28*

1 Announcements

- Midterm in class 10-12. Arrive by 9:45 am. Midterm grades will be given by percentage points, rather than by N/L/R grading.

2 Randomized Algorithms

Recommended Reading:

- CLRS Sec 9.0–9.2
- Roughgarden I Sec. 6.0–6.2
- 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 $[\texttt{var}_2] \in \{0, 1, \ldots \texttt{var}_2 - 1\}$.

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 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:

[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*.

- *Las Vegas Algorithms:* these are algorithms that always output a correct answer, but their running time depends on their random choices. In the worst case, Las Vegas algorithms can have infinite running time. Typically, we try to bound their *expected* running time. That is, we say the *(worst-case) expected running time* of A is where $T_A(x)$ 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 $\Pi = (\mathcal{I}, \mathcal{O}, f)$ *with error probability* p if

  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:

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 **:**

Computational Problem Selection

In particular, when $i = (n-1)/2$, we need to find a *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?

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
    $p = \texttt{random}(n)$;
    $P = K_p$ ;                                   /* $P$ 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                                  ;
    else if $i \geq n_< + n_=$ then                              ;
    else return $A_=[0]$;
```

Algorithm 1: QuickSelect

Example:

2. Proof of 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:
   $$T(n) \leq$$

□

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:

You can learn more about this conjecture in courses like CS121, CS221, and CS225.

2.4 Rigorous runtime analysis for QuickSelect

The detailed lecture notes have optional reading with a detailed proof of the runtime of QuickSelect.