CS120: Intro. to Algorithms and their Limitations Prof. Salil Vadhan

Lecture 8: Randomized Algorithms: QuickSelect

Harvard SEAS - Fall 2021 *Sept. 28, 2021*

1 Announcements

- Midterm in class 10/14
- PS5 due 10/20
- Final exam Fri 12/17, 9am
- Add/drop 10/4

Recommended Reading:

- Lewis-Zax Chs. 26-29 (especially 29)
- CLRS Sec 9.0–9.2
- Roughgarden I Sec. 6.0–6.2

2 Randomized Algorithms

One very useful ingredient to add to algorithms is *randomization*, where we allow the algorithm to generate "toss coins" or generate random numbers, and act differently depending on the results of those random coins. In the Word-RAM Model,: we can model randomization by introducing 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 127, 221, and 225 cover the theory of pseudorandom generators, 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:

- *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 that A has (worst-case) expected running time at most $T(n)$ if

$$\text{for every input } x \text{ of size } n,\ \mathrm{E}[\mathrm{Time}_A(x)] \leq T(n),$$

where $\text{Time}_A(x)$ is the random variable denoting the runtime of A on x, and $\text{E}[\cdot]$ denotes the expectation of a random variable:

$$\text{E}[Z] = \sum_{z \in \mathbb{N}} z \cdot \Pr[Z = z].$$

- *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 oh *all* inputs.

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.

3 QuickSelect

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

Input	: An array A of item-key pairs $((I_0, K_0), \ldots, (I_{n-1}, K_{n-1}))$, where each key $K_j \in \mathbb{N}$, and a *rank* $i \in [n]$
Output	: An item key-pair (I_j, K_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 — see Salil's course CS208 this spring if you are interested in 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 3.1. *There is a randomized algorithm QuickSelect that always solves Selection correctly, and has (worst-case) expected running time $O(n)$.*

Proof. 1. The algorithm:

```
1  QuickSelect(A, i)
   Input  : An array A = ((I_0, K_0), ..., (I_{n-1}, K_{n-1})), where each K_j ∈ ℕ, and i ∈ ℕ
   Output : An item key-pair (I_j, K_j) such that K_j is an i'th smallest key.
2  if n ≤ 1 then return (I_0, K_0);
3  else
4      p = random(n);
5      P = K_p ;                                   /* P is called the pivot */
6      Let A_< = an array containing the elements of A with keys < P;
7      Let A_> = an array containing the elements of A with keys > P;
8      Let A_= = an array containing the elements of A with keys = P;
9      Let n_<, n_>, n_= be the lengths of A_<, A_>, and A_= (so n_< + n_> + n_= = n);
10     if i < n_< then return QuickSelect(A_<, i);
11     else if i ≥ n_< + n_= then return QuickSelect(A_>, i − n_< − n_=) ;
12     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: Next time.

□