CS1200: Intro. to Algorithms and their Limitations Prof. Anurag Anshu

Lecture 9: Randomized algorithm

Harvard SEAS - Spring 2026 *2026-02-25*

1 Announcements

- SRE3 today.
- Pset 3 due tomorrow.
- Pset 4 out today.
- Anurag OH (after class).

2 Recommended Reading

- Hesterberg–Vadhan, Chapter 9.

3 Randomized algorithms: A motivating problem

A *median* of an array of n (potentially unsorted) key-value pairs $A = ((K_0, V_0), \ldots, (K_{n-1}, V_{n-1}))$ is the key-value pair (K_j, V_j) such that K_j is larger and smaller than at most $\lfloor (n-1)/2 \rfloor$ keys in the array.[1] There is much interest in algorithmic statistics, machine learning, and data privacy because of *robustness to outliers*.

The following computational problem generalizes the task of finding the median of an array of key-value pairs.

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 $(i+1)^{\text{st}}$- 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

We can solve Selection in time $O(n \log n)$, by Sorting (time $O(n \log n)$) and returning the i^{th} element of the sorted array (time $O(1)$). By introducing the power of *randomness*, we can obtain a simple and faster ($O(n)$) algorithm.

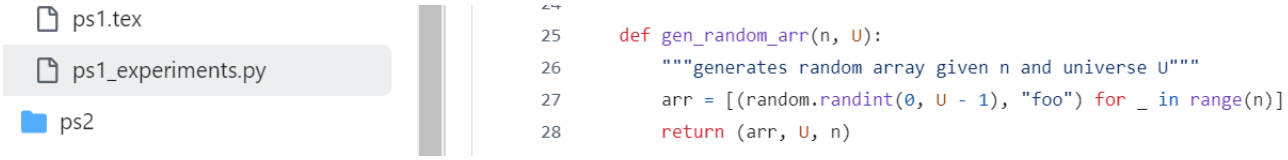

Figure 1: The random.randint command in Python allows you to generate a random integer from $[U]$.

4 QuickSelect

Recall the experiments code from Problem Set 1, shown in Figure 1.

This is an example of *randomization* in algorithms, 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*, which are outside the scope of this course.)

We prove the following theorem which gives the desired randomized algorithm for Selection. It is a Las Vegas algorithm.

Theorem 4.1. *There is a randomized algorithm, called* `QuickSelect`*, that always solves* Selection *correctly and has (worst-case) 'expected running time'* $O(n)$.

[1] If n is even or there are duplicate keys, there may be multiple pairs satisfying this definition, in which case each of them is called a median.

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 $(i+1)^{\text{st}}$-smallest key.
0 **if** $n \leq 1$ **then return** (K_0, V_0);
1 **else**
2 $p =$ `random`(n);
3 pivot $= K_p$;
4 Let $A_{\text{smaller}} =$
5 Let $A_{\text{larger}} =$
6 Let $A_{\text{equal}} =$
7 Let $n_{\text{smaller}}, n_{\text{larger}}, n_{\text{equal}}$ be the lengths of A_{smaller}, A_{larger}, and A_{equal} (so $n_{\text{smaller}} + n_{\text{larger}} + n_{\text{equal}} = n$);
8 **if** $i < n_{\text{smaller}}$ **then** ;
9 **else if** $i \geq n_{\text{smaller}} + n_{\text{equal}}$ **then** ;
10 **else return** $A_{\text{equal}}[0]$;

Algorithm 4.1: `QuickSelect()`

Example 4.2.

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

   $$\mathrm{E}\left[\max\{n_{\text{smaller}}, n_{\text{larger}}\}\right] \leq \frac{3n}{4}.$$

   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:

   $$T(n) \leq$$

□

5 Randomized Algorithms: Definitions

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. For some very unlikely random choices, Las Vegas algorithms are allowed to have very large running time. Typically, we try to bound their *expected* running time. QuickSelect is a Las Vegas algorithm.

- *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.

  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 and returning the most common answer. Here is a scenario where Monte Carlo algorithms are useful.

Example 5.1. A pollster is paid to estimate the percentage of the population that prefers to sing over dance, to within some small margin of error.

Input: An array A of people's opinions $(t_0, t_1, \ldots t_{n-1})$, where each $t_j \in \{S, D\}$, and a rational number $\varepsilon \in (0, 1)$.

Output:

Computational Problem ESTIMATE PROPORTION

A Monte Carlo algorithm to solve this problem is

Q: Which is preferable (Las Vegas or Monte Carlo)? That is, if someone offers to give you either a Las Vegas algorithm or a Monte Carlo algorithm solving a problem, which should you pick?

A: