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

Randomness in algorithms has been used for a long time, famously in Buffon's needle problem for estimating the value of π. But important changes are needed in our definition of an algorithm in order to bring in randomness.

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:

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

which assigns $\mathtt{var}_1$ a uniformly element of the set $[\mathtt{var}_2] \in \{0, 1, \ldots \mathtt{var}_2 - 1\}$.

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

$$T(n) = \max_{x:\text{size}(x)\leq n} \mathrm{E}[T_A(x)],$$

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:

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

Example: Consider the following computational problem.

Input	: A composite number n
Output	: A number $k < n$ that divides n.

<div align="center">

Computational Problem Factorize
</div>

A Las Vegas algorithm is as follows.

```
1 Factorize(n)
2 repeat = 1
3 sqrt = ⌊√n⌋
4 while repeat = 1 do
5     p = random(sqrt)
6     if p divides n then (repeat = 0 and return p);
```

The algorithm correctly solves the computational problem and its expected runtime is $O(\sqrt{n})$.

- *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

$$\text{for every } x \in \mathcal{I} \text{ s.t. } f(x) \neq \emptyset, \ \Pr[A(x) \in f(x)] \geq 1 - p, \text{ and}$$

$$\text{for every } x \in \mathcal{I} \text{ s.t. } f(x) = \emptyset, \ \Pr[A(x) = \bot)] \geq 1 - p$$

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). An example of Monte Carlo algorithm is the aforementioned Buffon's needle method.

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 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? 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:

```
1 QuickSelect(A, i)
  Input    : An array A = ((K₀, V₀), ..., (K_{n-1}, V_{n-1})), where each K_j ∈ ℕ, and i ∈ ℕ
  Output   : A key-value pair (K_j, V_j) such that K_j is an i'th smallest key.
2 if n ≤ 1 then return (K₀, V₀);
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: 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:

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

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

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

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

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 $n'_< \leq p'$ and $n'_> \leq n - p' - 1$, so we have the following (assuming n is odd for simplicity):

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

(The case of even n is similar.)

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

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

which would imply that

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

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 $T(n) \leq 4cn$ for all natural numbers $n \geq 1$ 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:

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

as desired. There are some statistical subtleties in the above steps that we have not expanded upon.

$\square$