CS1200: Intro. to Algorithms and their Limitations Anshu & Vadhan

Lecture 8: The Word-RAM Model and Randomized Algorithms

Harvard SEAS - Fall 2024 *2024-09-26*

1 Announcements

- PS0 regrade requests due today (before 11:59:59 PM)
- PS0 revision videos due Sunday (before 11:59:59 PM)
- Revision video guidelines linked on Ed
- Psets - remember to cite collaborators
- Salil OH SEC 3.327 after class, Anurag OH (Zoom) 1:30-2:30 pm

2 The Word-RAM Model

As noted above, an unrealistic feature of the RAM Model as we've defined it is it allows an algorithm to access and do arithmetic on arbitrarily large integers in one time step. In practice, the numbers stored in the registers of CPUs are of a modestly bounded *word length* w, e.g. $w = 64$ bits.

Q: How to represent and compute on larger numbers (e.g. multiplying two 1024-bit prime numbers when generating keys for the RSA public-key cryptosystem)?

A: Use "bignum" arithmetic, where we write our numbers as an array of digits in base 2^w (similarly to what you did in PS1 for Radix Sort with a large base b). So an n-bit number is now expressed as an array of $m = \lceil n/w \rceil$ words. Using the grade-school algorithm for multiplication, we can multiply two such numbers using $O(m^2)$ word-operations. There are asymptotically faster methods for multiplying large integers, such as Karatsuba's algorithm (taught in CS1240), which has runtime $O(m^{\log_2 3})$ and ones based on the Fast Fourier Transform that have runtime $O(m \log m)$.[1]

Using a finite word size leads us to the following computational model:

Definition 2.1. The *Word RAM Model* is defined like the RAM Model except that it has a *word length* w and *memory size* S that are used as follows:

- Memory: array of length S, with entries in $\{0, 1, \ldots, 2^w - 1\}$. Reads and writes to memory locations larger than S have no effect. Initially $S = n$, the length of the input array x. Additional memory can be allocated via a MALLOC command, which increments S by 1.

[1] Achieving this runtime is a development from just 2019, resolving a 40-year old conjecture! https://www.jstor.org/stable/10.4007/annals.2021.193.2.4

- Variables: In addition to the usual variables in a RAM Program, there is a variable `word_len` that is initialized to equal the word length w.
- Output: if the program halts, the output is defined to be $(M[\texttt{output\_ptr}], M[\texttt{output\_ptr}+1], \ldots, M[\min\{\texttt{output\_ptr}+\texttt{output\_len}-1, S-1\}])$. That is, portions of the output outside allocated memory are ignored.
- Operations: Addition and multiplication are redefined from RAM Model to return $2^w - 1$ if the result would be $\geq 2^w$.[2]
- Crashing: A Word-RAM program P *crashes* on input x and word length w if any of the following occur:
  1. One of the constants c in the assignment commands ($\texttt{var} = c$) in P is $\geq 2^w$.
  2. $x[i] \geq 2^w$ for some $i \in [n]$
  3. $S > 2^w$. (This can happen because $n > 2^w$, or if $2^w - n + 1$ MALLOC commands are executed.)

We denote the computation of a Word-RAM program on input x with word length w by $P[w](x)$. Note that $P[w](x)$ has one of three outcomes:

- halting with an output
- failing to halt, or
- crashing.

We define the *runtime* $T_{P[w]}(x)$ to be the number of commands executed until P either halts or crashes (so $T_{P[w]}(x) = \infty$ if $P[w](x)$ fails to halt).

This model, with say $w = 32$ or $w = 64$, is a reasonably good model for modern-day computers with 32-bit or 64-bit CPUs.

Q: What's wrong with just fixing $w = 64$ in Definition 2.1 and using it as our model of computation?

A: If we do this, then our RAM programs can only access at most 2^w memory locations, for a total of at most $w \cdot 2^w$ bits of information. Thus, they cannot solve even simple problems like addition on arbitrarily large inputs; this goes counter to our intuitive notion of algorithm, which is a fixed procedure that solves a problem on arbitrarily large inputs.

Thus, we instead keep w as a varying parameter in Definition 2.1, which gives us a single program P that can be instantiated with different word lengths in order to handle arbitrarily large inputs (or computations that access arbitrarily large amounts of memory). We need to take care for how we define what it means for a Word-RAM program to solve a computational problem, and how we definite its runtime:

[2] A more standard choice is for the result to be returned mod 2^w, but we instead clamp results to the interval $[0, 2^w - 1]$ (known as *saturation arithmetic*) for consistency with how we defined subtraction in the RAM Model. If all arithmetic is done modulo 2^w, then the Conditional GOTO should also be modified to allow the condition to be an inequality, since we can no longer use the subtraction to simulate inequality tests.

Definition 2.2. We say that word-RAM program P *solves* computational problem $\Pi = (\mathcal{I}, \mathcal{O}, f)$ if the following holds for every input x,

1. There is at least one word length $w \in \mathbb{N}$ such that $P[w](x)$ halts without crashing.
2. For every word length $w \in \mathbb{N}$ such that $P[w](x)$ halts without crashing, the output $P[w](x)$ satisfies $P[w](x) \in f(x)$ if $f(x) \neq \emptyset$ and $P[w](x) = \perp$ if $f(x) = \emptyset$.

A mental model for this definition is that if we find that our current computer's word size w is not sufficient to complete the execution of $P[w](x)$ without crashing, then we can go out and buy (or build) better hardware with a larger word size, until we find one that completes. Since larger word size generally means a more powerful computer, we expect the running time to decrease as the word-size gets larger, but we'll often want to run our program with whatever hardware we have in hand, or the smallest word size we can. Thus, it makes sense to measure complexity as follows:

Definition 2.3. The *running time* of a word-RAM program P on an input x is defined to be

$$T_P(x) = \max_{w \in \mathbb{N}} T_{P[w]}(x).$$

Like in Lecture 2, we define the worst-case running time of P to be the function $T_P(n)$ that is the maximum of $T_P(x)$ over inputs x of size at most n.

In many algorithms texts, you'll see the word size constrained to be $O(\log n)$, where n is the length of the input. This is justified by the following:

Proposition 2.4. *For a word-RAM program P and an input x that is an array of n numbers, if $T_P(x) < \infty$, then there is a word size w_0 such that $P[w](x)$ does not crash for any $w \geq w_0$. Specifically, we can take*

$$w_0 = \lceil \log_2 \max \{n + T_P(x), x[0], \ldots, x[n-1], c_0, \ldots, c_{k-1}\} + 1 \rceil,$$

where $c_0, \ldots, c_{k-1}$ are the constants appearing in variable assignments in P.

This proposition implies that if the runtime $T_P(x)$ is at most $\text{poly}(n)$ and the numbers in x are of magnitude at most $\text{poly}(n)$, the we need word size at most $w_0 = O(\log n)$ to support the computation. (The constants $c_0, \ldots, c_{k-1}$ are just constants and have no dependence on n.) Most algorithms courses focus on algorithms that run in time $\text{poly}(n)$, so there is no harm in restricting to word length $O(\log n)$. However, in CS1200, we will sometimes study algorithms that run in exponential time or don't even halt, and thus may also use much more than $\text{poly}(n)$ memory locations.

Proof. The setting of w_0 and $w \geq w_0$ ensures that

$$2^w \geq 2^{w_0} \geq \max \{n + T_P(x), x[0], \ldots, x[n-1], c_0, \ldots, c_{k-1}\} + 1.$$

In an execution of $P[w](x)$, the memory size S never exceeds $n + T_P(x)$, because it starts at n and grows by at most one in each time step (which happens only if a MALLOC command is executed). Thus we are guaranteed to maintain $S \leq 2^w - 1$, so that the program won't crash because of allocating too much memory. Since $x[0], \ldots, x[n-1] \leq 2^w - 1$, $P[w](x)$ will not crash because of values assigned to memory locations. Since $c_0, \ldots, c_{k-1} \leq 2^w - 1$, then $P[w](x)$ will not crash because of variable assignments. □

Different algorithms researchers allow different sets of basic operations in the Word-RAM Model, just like different CPUs have different instruction sets. Some of these make only a constant-factor difference in running time (like the % example we discussed earlier), but some may make a difference that depends polynomially on the word length w. For example, it is not obvious how to implement the bitwise XOR of two w-bit words using a constant number of operations in the Word-RAM model we defined, but it can be done using $O(w)$ operations, which usually translates to a logarithmic factor in runtime. It turns out that allowing such bitwise operations, it is possible to sort integers in time $O(n \cdot \log\log n)$, with no dependence on the size of the key universe $[U]$, beating both MergeSort and RadixSort for large universe sizes (specifically, when $\log U = \omega(\log n \cdot \log\log n)$).

Overall, the Word-RAM Model has been the dominant model for analysis of algorithms for over 50 years, and thus has stood the test of time even as computing technology has evolved dramatically. It still provides a fairly accurate way of measuring how the efficiency of algorithms scales. That said, it is worth noting a few of the ways in which one can observe real-world performance that deviates from the Word-RAM model's predictions:

1. Not all operations take the same amount of time. As discussed above, these differences typically lead to constant or logarithmic factors in runtime, which may be noticeable in practice.

2. The memory hierarchy. In real-life computers, not all memory accesses are equivalent. Reading from registers vs. cache vs. main memory vs. disk all take significantly different amounts of time. It is possible to define computational models that account for the memory hierarchy, but they are beyond the scope of this course.

3. Parallelism. The RAM model captures only a single CPU operating sequentially, and doesn't model the performance we can gain from parallel computers (e.g. multi-core machines) or distributed computation. There are models for parallel and distributed algorithms, but also beyond the scope of this course.

Ignoring some of these aspects of computing technology is the price we pay for having a mathematically clean general-purpose theory of algorithms that lasts through changes in technology.

3 Word-RAM vs. RAM

Although the Word-RAM Model and the RAM Model each have their advantages (the Word-RAM is more realistic, while RAM is simpler), we can simulate each one by the other.

Theorem 3.1. *1. For every RAM program P, there is a Word-RAM Program P' that simulates P. That is, for every input x, $T_{P'}(x) = \infty$ iff $T_P(x) = \infty$, and if both are finite, then for every word size w such that $P'[w](x)$ halts without crashing (this holds for all sufficiently large w by Proposition 2.4), the output of $P'[w](x)$ is equal to (an encoding of) the output of $P(x)$. Furthermore,*

$$T_{P'[w]}(x) = O\left((T_P(x) + n + S) \cdot \left(\frac{\log M}{w}\right)^{O(1)}\right), \tag{1}$$

where n is the length of the input x, S is the largest memory location accessed by P on input x,[3] *and M is the largest number computed by P on input x.*

2. *For every Word-RAM program P, there is a RAM program P' that simulates P in the sense that P' halts on (w, x) iff $P[w]$ halts on x, and if they halt, then the output of $P'(w, x)$ equals the output of $P[w](x)$ or* `crash` *according to whether $P[w](x)$ crashes or not. Furthermore,*

$$T_{P'}(w, x) = O\left(T_{P[w]}(x) + n + w\right),$$

where n is the length of x.

The runtime bound (1) in Part 1 says that RAM programs can be simulated by Word-RAM programs with a slowdown that depends on the magnitude M of numbers computed by the RAM program. For example, if the RAM program only computes numbers of magnitude $M = n^{O(1)}$, then the there is only a constant-factor slowdown, since the smallest non-crashing word length w must be at least $\log n$ (in order for the memory to fit the input) and $\log(n^{O(1)})/\log n = O(1)$. However, RAM programs can compute very large numbers (cf. PS3), so in general the slowdown can be quite substantial. Note also that the right-hand size of (1) is maximized at $w = 1$, so this that gives us a bound on $T'_P(x) = \max_w T_{P'[w]}(x)$, which can be plugged into Proposition 2.4 to determine an upper bound on the word size needed to ensure that $P'[w](x)$ does not crash.

In Part 2, we give the word size w of P to be simulated to P' as input. To obtain a simulating RAM program P'' that is just given x, we can try running $P'(x, w)$ with $w = 1, 2, 3, \ldots$ until we find a word-size where $P'(x, w)$ does not output `crash`. We know by Proposition that if $T_{P'}(x) < \infty$, then this search will stop no later than a word size w_0, which is at most logarithmic in $T_{P'}(x)$ and thus has a small cost in runtime. Moreover, if we can compute an upper bound on w_0 efficiently, then we can just use that bound as word size to avoid searching over word sizes w.

Proof Sketch. 1. Our Word-RAM program will simulate the execution of the RAM program, one line at a time. We need to find a way to simulate anything the RAM program does that's not directly part of the Word-RAM model. Our Word-RAM simulation will then replace each number stored by P using a bignum representation (stored as a linked list of digits in memory) in base 2^w. It will also periodically use MALLOC commands to ensure that the memory remains large enough for P's memory (which will have an additive cost of $O(S)$ in runtime) and for the bignum representations of numbers (which will incur a multiplicative blow-up of $O(\log M)/w$ since this upper bounds the number of digits to represent numbers from $[M]$ in base 2^w.). Operations will be simulated using bignum arithmetic, which incurs a multiplicative runtime blowup of $O((\log M)/w)^2$ if we use the grade-school algorithms for multiplication and division.

2. Problem Set 3.

□

[3] Any RAM Program can be modified so that $S = O(n + T_P(x))$ by using a Dictionary data structure, where the keys represent memory locations and the values represent numbers to be stored in those locations. The total number of elements to be stored in the data structure is bounded by n (the initial number of elements to be stored) and plus the number of writes that P performs, which is at most $T_P(x)$. Depending on whether the Dictionary data structure is implemented using Balanced BSTs or Hash Tables, this transformation may incur a logarithmic-factor blow-up in runtime or yield a Las Vegas randomized algorithm.

4 Takeaway

The Word-RAM model is the formal model of computation underlying everything we are doing in CS1200. Because of Proposition 2.4, we assume the word size is $w = O(\log(n + T(n)))$ for algorithms that run in time $T(n)$. After today, we will return to writing high-level pseudocode and won't torture ourselves with continuously writing Word-RAM code, but implicitly all of our theorems are about the Word-RAM programs that would be obtained by compiling our pseudocode into Word-RAM form. Thus, the Word-RAM model is the reference to use when we need to figure about how long some operation would take.

We usually won't need to worry too much about the distinction between the RAM Model and the Word RAM Model, since the numbers involved and memory usage of our algorithms will typically be polynomial in n, so can all be managed with a word length of $O(\log n)$. But it's worth keeping in the back of your mind: if it looks like your algorithm might construct numbers that are much larger than the input size, then we need to pay closer attention and not treat arithmetic operations as constant time.

5 Randomized Algorithms

Recommended Reading:

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

5.1 A motivating problem

The *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. 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 (average value of the keys). 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).

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 $i+1$st smallest key. That is, there are at most i possible k such that $K_k < K_j$ and there are at most $n - i - 1$ possible k such that $K_k > K_j$.

Computational Problem Selection

We can solve Selection in time $O(n \log n)$ for any i, by Sorting (time $O(n \log n)$) and returning the i'th element of the sorted array (time $O(1)$). But by introducing the power of "randomness", we can obtain a simpler and faster algorithm.

5.2 Definitions

Recall that one criterion making the RAM and Word-RAM models 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. For instance, in Pset1 (Problem 3(e)), you may have noticed different scatter plots when running the program multiple times. In turns out that we had included the lines of codes depicted in Figure 1.

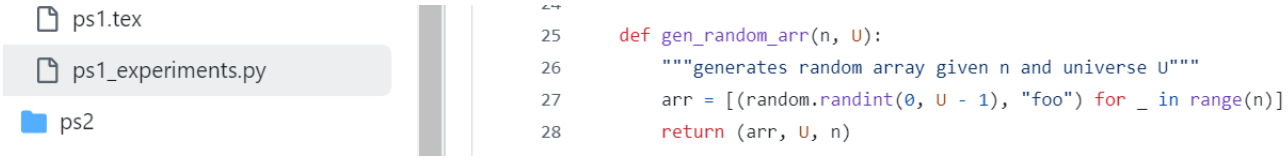

Figure 1: The random.randint command in Python allows you to generate a random integer from $[U]$.

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*. Courses such as CS 1210, 1270, 2210, and 2250 cover the theory of pseudorandom generators,[4] 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. 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. That is, we say the *(worst-case) expected running time* of A is

$$T_{expected}(n) = \max_{x:\mathsf{size}(x) \leq n} \mathrm{E}[T_A(x)],$$

[4] 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*.

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:

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

We will soon see an example of a Las Vegas algorithm for solving the Selection problem (Theorem 5.1).

- *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 the following holds for every $x \in \mathcal{I}$,

  1. $\Pr[A(x) \in f(x)] \geq 1 - p$ if $f(x) \neq \emptyset$ and
  2. $\Pr[A(x) = \perp] \geq 1 - p$ if $f(x) = \emptyset$.

  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). Here is a scenario where Monte Carlo algorithms are useful.

  Example: Our course coordinators, Allison and Emma, are excited to help out with Fall 24's CS 1200 T-shirt orders. After receiving the order from the vendor, they want to double check that the sizes of T-shirts are roughly in the right number. However, the vendor mixed up S and M sizes and put them in one box (L sizes are in a separate box). Allison and Emma need to quickly estimate the number of S and M sizes, and are faced with the following computational problem:

Input	**:** An array A of t-shirt labels $(t_0, t_1, \ldots t_{n-1})$, where each $t_j \in \{S, M\}$, and a rational number $\varepsilon \in (0, 1)$
Output	**:** A rational number $r \in (0, 1)$ which is a good estimate of the fraction of S-sized t-shirts. That is $\lvert r - \lvert\{i : t_i = S\}\rvert / n\rvert \leq \varepsilon.$

Computational Problem EstimateSize

  A Monte Carlo algorithm to solve this problem is to look at $O(\lceil 1/\varepsilon^2 \rceil)$ random indices and and output the fraction of $t'_j s$ which are equal to S among those indices. With probability ≥ 0.99, the output is close to the actual fraction of S-sized shirts. This algorithm is described in Section 5.6, which is an optional reading material.

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, there are problems for which Monte Carlo algorithms achieve better runtime, such as for the problem EstimateSize. Testing whether a large (bignum) integer is prime is another example where we have Monte Carlo algorithms that are faster than any known Las Vegas algorithms.

5.3 QuickSelect

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

Theorem 5.1. *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 = ((K_0, V_0), ..., (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 + 1st smallest key.
2  if n ≤ 1 then return (K_0, V_0);
3  else
4      p = random(n);
5      Pivot = K_p;
6      Let A_smaller = an array containing the elements of A with keys < Pivot;
7      Let A_larger = an array containing the elements of A with keys > Pivot;
8      Let A_equal = an array containing the elements of A with keys = Pivot;
9      Let n_smaller, n_larger, n_equal be the lengths of A_smaller, A_larger, and A_equal (so
         n_smaller + n_larger + n_equal = n);
10     if i < n_smaller then return QuickSelect(A_smaller, i);
11     else if i ≥ n_smaller + n_equal then return QuickSelect(A_larger, i − n_smaller − n_equal) ;
12     else return A_equal[0];
```

Algorithm 1: QuickSelect

Example: Let $A = [2, 3, 5, 6, 3, 5, 4]$ with $n = 7$ and $i = 3$ (here we only keep track of the keys). Note that the 4th smallest key is 4.

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

2. **Proof of correctness sketch:**
The proof uses induction on size of the arrays. In particular, we work with the following inductive hypothesis P_k :

"For all A arrays of size k, `QuickSelect(`A,i`)` returns the $i + 1$st smallest key."

The base case P_1 holds due to Line 2. For the inductive step, we assume that P_k holds for all $k < n$. Then we show P_n by inspecting Lines 10 and 11 and observing that QuickSelect is recursively called on a smaller array with the right index (i in Line 10 and $i - n_{\text{smaller}} - n_{\text{larger}}$ in Line 11).

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 5.5 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 $\max\{n_{\text{smaller}}, n_{\text{larger}}\}$. 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:

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

□

5.4 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: Theorem 5.1 gave a 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.

5.5 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:

$$\mathrm{E}\left[T_{\text{QuickSelect}}(A)\right] = \mathrm{E}_p\left[\mathrm{E}\left[T_{\text{QuickSelect}}(A)|p\right]\right], \tag{2}$$

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

$$\mathrm{E}_p\left[T_{\text{QuickSelect}}(A)|p\right] \leq cn + \max\{T(n_{\text{smaller}}), T(n_{\text{larger}})\} = cn + T(\max\{n_{\text{smaller}}, n_{\text{larger}}\}). \tag{3}$$

$$\mathrm{E}_p[\max\{n_{\text{smaller}}, n_{\text{larger}}\}] \leq 3n/4, \tag{4}$$

Equation (2) is a general probability fact known as the Law of Iterated Expectations. Inequality (3) follows from inspection of the algorithm.

To prove Inequality (4), we observe that the lengths $n_{\text{smaller}}, n_{\text{larger}}, n_{\text{equal}}$ 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 $n'_{\text{smaller}}, n'_{\text{larger}}, n'_{\text{equal}}$ of the corresponding partition of the sorted array has the same distribution as $n_{\text{smaller}}, n_{\text{larger}}, n_{\text{equal}}$.

With this choice, we observe that $n'_{\text{smaller}} \leq p'$ and $n'_{\text{larger}} \leq n - p' - 1$, so we have the following (assuming n is odd for simplicity):

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

(The case of even n is similar.)

Equations/Inequalities (2), (3), and (4) 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

$$\mathrm{E}[f(X)] = f(\mathrm{E}[X]) \tag{5}$$

(We are interested in $f = T$ and $X = \max\{n_{\text{smaller}}, n_{\text{larger}}\}$.) However, this Equation (5) 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:

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

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

□

5.6 Monte Carlo algorithm for EstimateSize

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

The algorithm is as follows.

```
1 RandomSubsampling(A, ε)
  Input  : An array A = (t_0, t_1, ... t_{n-1}), where each t_j ∈ {S, M}, and ε ∈ (0, 1)
  Output : A rational number r such that |r − |{i : t_i = S}|/n| ≤ ε
2 L = 100⌈1/ε²⌉ ;
3 small = 0;
4 foreach ℓ = 0, ... L − 1 do
5 |   j_ℓ = random(n);
6 |   if t_{j_ℓ} = S then
7 |   |   small = small + 1
8 r = small/L;
9 return r
```

Algorithm 2: RandomSubsampling

Line 5 samples a random index $j_\ell \in [n]$ and Line 6 checks if $t_{j_\ell} = S$. The variable small counts the number of S in the sampled coordinates. Note that the whole algorithm takes $O(L) = O(\lceil \frac{1}{\varepsilon^2} \rceil)$ time.

For correctness, we introduce a random variable called the indicator function: $\mathrm{Ind}(t_{j_\ell} = S)$ is 1 if $t_{j_\ell} = S$ and 0 otherwise. Thus, it captures if the If statement in Line 6 goes through. Note that

it is a random variable as it depends on j_ℓ which is randomly chosen in Line 5. We can write

$$r = \frac{\text{small}}{L} = \frac{\sum_{\ell=0}^{L-1} \text{Ind}(t_{j_\ell} = S)}{L}.$$

As defined, r is also a random variable, as it depends on $j_0, \ldots j_{L-1}$ which are all randomly chosen through executions in Line 5. We compute the expected value of r and the variance about the expected value, using the linearity of expectations:

$$\text{E}(r) = \frac{\sum_{\ell=0}^{L-1} \text{E}(\text{Ind}(t_{j_\ell} = S))}{L} = \frac{\sum_{\ell=0}^{L-1} \Pr(t_{j_\ell} = S)}{L} = f_S,$$

(where $f_S = |\{i : t_i = S\}|/n$) and

$$\begin{aligned}
\text{E}(r^2) - \text{E}(r)^2 &= \frac{\sum_{\ell,\ell'=0}^{L-1} \text{E}(\text{Ind}(t_{j_\ell} = S) \cdot \text{Ind}(t_{j_{\ell'}} = S))}{L^2} - f_S^2 \\
&= \frac{\sum_{\ell=0}^{L-1} \text{E}(\text{Ind}(t_{j_\ell} = S)}{L^2} + \frac{\sum_{\ell \neq \ell'} \text{E}(\text{Ind}(t_{j_\ell} = S) \cdot \text{Ind}(t_{j_{\ell'}} = S))}{L^2} - f_S^2 \\
&= \frac{\sum_{\ell=0}^{L-1} \Pr(t_{j_\ell} = S)}{L^2} + \frac{\sum_{\ell \neq \ell'} \Pr(t_{j_\ell} = S) \cdot \Pr(t_{j_{\ell'}} = S)}{L^2} - f_S^2 \\
&= \frac{f_S}{L} + \left(1 - \frac{1}{L}\right) f_S^2 - f_S^2 \leq \frac{f_S}{L},
\end{aligned}$$

where the second line divided the sum over ℓ, ℓ' in two cases: $\ell = \ell'$ or $\ell \neq \ell'$ and the third line uses the independence of random samples. The variance is often denoted as $\sigma^2 = \text{E}(r^2) - \text{E}(r)^2$. By Chebyshev's inequality, we find that

$$\Pr(|r - f_S| \geq \varepsilon) = \Pr(|r - f_S| \geq \frac{\varepsilon}{\sigma}\sigma) \leq \frac{\sigma^2}{\varepsilon^2} = \frac{f_S}{L\varepsilon^2} \leq \frac{1}{L\varepsilon^2}.$$

With the choice of L, this is upper bounded by 0.01.