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

- Regrade requests due today (before 11:59:59 PM)
- 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:

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:
- Output:

- Operations:
- Variables:
- Crashing: A Word-RAM program *crashes* on input x and word length w if any of the following happen during its computation:

  1.
  2.

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:

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:

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 $f(x) = \emptyset$.

Mental model: if $P[w](x)$ crashes, buy a better computer with a larger word size w and try again. 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

$$T_P(x) =$$

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.

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.

Q: What's not captured by the Word-RAM model?

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 (which exists by Proposition 2.4), the output of $P'[w](x)$ is equal to (an encoding of) the output of $P(x)$. Furthermore,*

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

where n is the length of the input x, S is the largest memory location accessed by P on input x,[1] and M is the largest number computed by P on input x.

[1] 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.

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.

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 :

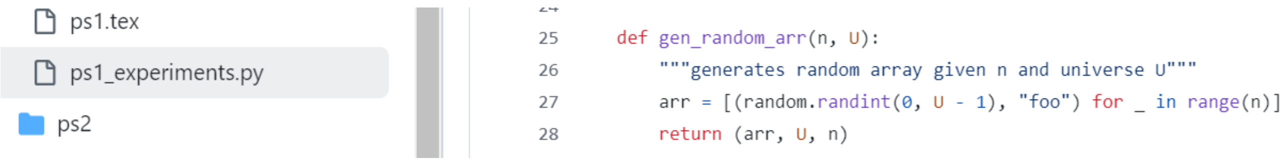
Computational Problem Selection

We can solve Selection in time $O(n \log n)$ for any i,

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.



```
def gen_random_arr(n, U):
    """generates random array given n and universe U"""
    arr = [(random.randint(0, U - 1), "foo") for _ in range(n)]
    return (arr, U, n)
```

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 121, 127, 221, and 225 cover the theory of pseudorandom generators,[2] 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:

[2] 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. 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.

- *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). 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 an $\varepsilon \in (0, 1)$

Output **:**

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'_js which are equal to S among those indices. With probability ≥ 0.99, the output is close to the actual fraction of S-sized shirts.

Q: Which is preferable (Las Vegas or Monte Carlo)?

A:

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:

```
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 = $ random($n$);
    Pivot $= K_p$;
    Let $A_{\text{smaller}} =$
    Let $A_{\text{larger}} =$
    Let $A_{\text{equal}} =$
    Let $n_{\text{smaller}}, n_{\text{larger}}, n_{\text{equal}}$ be the lengths of $A_{\text{smaller}}$, $A_{\text{larger}}$, and $A_{\text{equal}}$ (so
      $n_{\text{smaller}} + n_{\text{larger}} + n_{\text{equal}} = n$);
    if $i < n_{\text{smaller}}$ then                              ;
    else if $i \geq n_{\text{smaller}} + n_{\text{equal}}$ then                              ;
    else return $A_{\text{equal}}[0]$;
```

Algorithm 1: QuickSelect

Example:

2. **Proof of correctness:**

3. **Expected runtime:**

   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:

   $$T(n) \leq$$

□

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:

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