CS1200: Intro. to Algorithms and their Limitations Prof. Salil Vadhan

Lecture 9: The Extended CT Thesis and Randomized Algorithms

Harvard SEAS - Fall 2025 *2025-09-30*

1 Announcements

- Salil's Thursday OH this week moved to 12:30pm-1:15pm, SEC 3.327.
- SRE 3 moved to Thursday 10/9.

Recommended Reading:

- Hesterberg–Vadhan Ch. 10
- MacCormick Ch. 5, 10
- CLRS Sec 9.0–9.2
- Roughgarden I Sec. 6.0–6.2
- Lewis-Zax Chs. 26-29

2 Loose Ends: 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. Moreover, the simulations preserve runtime if we restrict to RAM programs that don't utilize large numbers or access far-away memory.

Theorem .1. *Let $\Pi = (\mathcal{I}, \mathcal{O}, f)$ be a computational problem, with some fixed encoding of elements of $\mathcal{I}$ as arrays of natural numbers. Let $T : \mathcal{I} \to \mathbb{R}^+ \cup \{\infty\}$ such that for all $x \in \mathcal{I}$, $T(x) \geq n+2$,[1] where n as the length of the array encoding x, and every entry of that array has bitlength $O(\log T(x))$. Then the following are equivalent:*

1. *Π can be solved by a Word-RAM program P with $T_P(x) = O(T(x))$.*
2. *Π can be solved by a RAM program Q with $T_Q(x) =$ that also satisfies the following conditions for all input arrays x:*

   (a)

[1] The $+2$ is just to ensure that $\log T(x) \geq 1$ even when $n = 0$.

(b)

As a corollary (taking $T(x)$ to be arbitrarily large but finite for each $x \in \mathcal{I}$), we deduce that Word-RAM is a Turing-equivalent model:

Corollary .2. *A computational problem can be solved by a Word-RAM program if and only if it can be solved by a RAM program.*

Note that the above theorem refers to a fixed encoding of inputs as arrays of natural numbers, which we use for both the RAM and Word-RAM programs. For problems involving numbers, there is an important distinction between

- *smallnum* problems:
- *bignum* problems:

All of the algorithms we have seen (like `ExhaustiveSearchSort`, `InsertionSort`, `MergeSort`, `SingletonBucketSort`, `IntervalSchedulingViaSorting`, `RadixSort`), if applied to problems, can be implemented by RAM programs running in the time bounds $O(T)$ that we have claimed (e.g. $T(n) = n \cdot n!$, $T(n) = n^2$, $T(n) = n \cdot \log n$, $T(n, U) = O(n + U)$, and $T(n, U) = O(n + n \cdot (\log U)/(\log n))$) satisfying the additional conditions of only working with numbers of bitlength $O(\log T)$ and using at most the first $O(T)$ memory locations. Thus we deduce from Theorem .1 that their runtimes also hold for the Word-RAM model.

Proof Sketch of Theorem .1.

(1)⇒(2). Mostly done by Problem Set 2. See textbook for how to deduce the claim from what you do on the problem set.

(2)⇒(1). Idea: start with $O(T(x))$ MALLOC operations to ensure that (a) we have enough space for memory used by Q, and (b) the word size is at least $\log_2 T(x)$. The latter implies that each number used by Q fits in $O(1)$ words of P and we slowdown only by a constant factor:

Additional technicality: we don't know $T(x)$ in advance, so we try time bounds of $t = 1, 2, 4, 8, 16$ until we succeed in the simulation. □

The constraint that the RAM program only manipulates numbers of bitlength $O(\log T(x))$ is essential for an efficient simulation by Word-RAM programs. If the RAM program instead computes numbers of bitlength up to $B(x)$, then the bignum arithmetic in the simulation would incur a slowdown factor of $(B(x)/(\log T(x)))^{O(1)}$.

3 The Extended (or Strong) Church–Turing Thesis

The Church–Turing Thesis only concerns problems solvable at all by these models of computation (Word-RAM programs, etc.). We haven't even seen any problems that are *not* solvable by Word-RAM programs—that will be a topic for the end of the course. There is, however, a stronger version of the Church–Turing Thesis that also covers the efficiency with which we can solve problems:

Extended Church–Turing Thesis v1:

The Extended Church–Turing Thesis is not a precise mathematical claim, and thus cannot be formally proven. In fact, randomized algorithms, massively parallel computers, and quantum computers all could potentially provide an exponential savings in runtime.

If we modify the statement with some qualifiers, then these challenges no longer apply:

Extended Church–Turing Thesis v2:

This form of the Extended Church–Turing Thesis has stood the test of time for the approximately fifty years since it was formulated, even as computing technology has changed tremendously in that time.

Like the basic Church–Turing Thesis, the Extended Church–Turing Thesis was originally formulated for Turing Machines, as discussed in the textbook. Turing Machines and Word-RAM Programs are *polynomially equivalent*, meaning that they can simulate each other with a polynomial slowdown. Any computational model that is polynomially equivalent to these is referred to as a *strongly Turing-equivalent* model of computation.

4 Word-RAM Takeaway

The Word-RAM model is the formal model of computation underlying everything we are doing in this course. But it is usually more convenient to use the equivalence with a contrained RAM model as given by Theorem .1, meaning that in addition

to measuring the runtime, we also need to confirm that the algorithms do not manipulate very large numbers (or if they do, take into account the time for bignum arithmetic).

We will return to writing high-level pseudocode, but these models are the reference to use when we need to figure about how long some operation would take.

5 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.[2] Of 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:

Computational Problem SELECTION

We can solve SELECTION in time _____ but by introducing the power of *randomness*, we can obtain a simple and faster algorithm.

6 Randomized Algorithms: Definitions

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

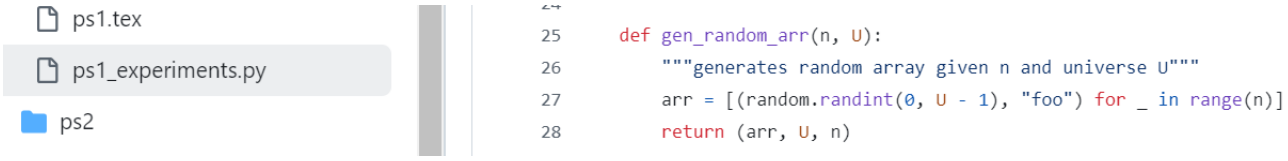

Figure 1: The random.randint command in Python allows you to generate a random integer from $[U]$.

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

[2] 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.

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.)

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.

- *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 .3. A pollster is paid to estimate the percentage of the population of the Dakota Territory that supports statehood, to within some small margin of error. The 19^{th} century was before the dawn of spam, so everyone will answer if asked, but the pollster would like to save effort by asking as few people as possible.

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

Output:

Computational Problem EstimateProportion

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:

7 QuickSelect

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

Theorem .4. *There is a randomized algorithm, called* `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), ..., (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+1)^st-smallest key.
 0 if n ≤ 1 then return (K_0, V_0);
 1 else
 2     p = random(n);
 3     pivot = K_p;
 4     Let A_smaller =
 5     Let A_larger =
 6     Let A_equal =
 7     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);
 8     if i < n_smaller then                                              ;
 9     else if i ≥ n_smaller + n_equal then
         ;
10     else return A_equal[0];
```

Algorithm .1: `QuickSelect()`

Example .5.

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:

   $$T(n) \leq$$

□

8 The Power of Randomized Algorithms

A fundamental question is whether allowing randomization (in either the Las Vegas or Monte Carlo ways) actually adds power: are there problems with faster randomized solutions than the fastest deterministic ones? Here are some examples of problems that reflect a potential gap between randomized and deterministic algorithms:

- Selection: Theorem .4 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 is a constant factor slower).
- Primality Testing: Given a bignum integer of size n words of memory, check if it is prime. There is an $O(n^3)$-time Monte Carlo algorithm, $O(n^4)$-time Las Vegas algorithm, and a $O(n^6)$-time deterministic algorithm (proven in the paper "Primes is in P").
- Identity Testing: Given some arithmetic expression (like $12875^{8091761235} \cdot (15676 + 91247^{259}) - 967012^{8107069871}$), check if it is equal to zero. This has an $O(n^2)$-time Monte Carlo algorithm, and the best currently-known deterministic algorithm runs in $2^{O(n)}$!

Nevertheless, the prevailing conjecture is:

You can learn more about this conjecture in courses like CS1210, CS2210, and CS2253.