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

Lecture 7: RAM vs Word-RAM, and Church-Turing thesis

Harvard SEAS - Spring 2026 *2026-02-23*

1 Announcements

- SRE3 on Wednesday

2 Recommended Reading

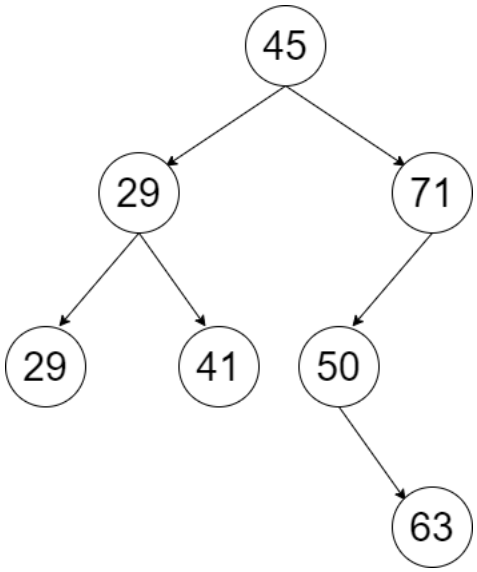
- Hesterberg–Vadhan, 7.2-7.4, 9.1-9.2.

3 Data

Implicit in the 'expressivity' requirement is that we can describe the inputs and outputs of algorithms in the model. In the RAM model, all inputs and outputs are arrays of natural numbers. How can we represent other types of data?

- (Signed) integers:
- Rational numbers:
- Real numbers:
- Strings:

What about a fancy data structure like a binary search tree? We can represent a BST as an array of 4-tuples (K, V, P_L, P_R) where P_L and P_R are pointers to the left and right children. Let's consider the example:



Assuming all of the associated values are 0, this would be represented as the following array of length 28:

$$[45, 0, 4, 8, 29, 0, 12, 16, 71, 0, 20, 0, 29, 0, 0, 0, 41, 0, 0, 0, 50, 0, 0, 24, 63, 0, 0, 0]$$

For nodes that do not have a left or right child, we assign the value of 0 to P_L or P_R. Assigning the pointer value to 0 does not refer to the value at the memory location of 0, as we assume that the root of the tree cannot be a child of any of the nodes in the rest of the tree. Note that there are many ways to construct a binary tree using this array representation.

4 Recap: Word-RAM model

Definition 4.1. The *Word-RAM program* P is defined like a RAM program except that it has a (static) word length parameter w and a (dynamic) *memory size* S.

- The word length w is *static*, meaning that while it can be set to an arbitrary value in $\mathbb{N}$ prior to running P on an input x, it does not change during the computation.
- The memory size S is *dynamic*, meaning that it can change during the computation, as described below.

These are used as follows:

- **Memory**: the memory is an array of length S, with entries in $\{0, 1, \ldots, 2^w - 1\}$. Reads from 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.
- **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 $2^w - 1$ (the max possible value) if the result would be $\geq 2^w$.
- **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).

Definition 4.2. We say that a word-RAM program P *solves* a computational problem $\Pi = (\mathcal{I}, \mathcal{O}, f)$ if the following hold for every input $x \in \mathcal{I}$:

1.
2.
3.

Definition 4.3. The *running time* of a word-RAM program P on an input x is defined to be

$$T_P(x) =$$

5 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 5.1. *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)
   (b)

As a corollary (taking $T(x)$ to be arbitrarily large but finite for each $x \in \mathcal{I}$), we deduce that

Corollary 5.2. *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

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

- *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 5.1 that their runtimes also hold for the Word-RAM model.

Proof Sketch of Theorem 5.1.

(1)⇒(2). Mostly done by Problem Set 2. See textbook (Page 84) 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 $(B(x)/(\log T(x)))^{O(1)}$.

6 The Church–Turing Thesis: Simulating the Universe

As a consequence of the simulation Theorems, and expanding it to include a few other models, we see that many different models of computation are equivalent in power, meaning that they can solve the same class of computational problems. We refer to such models of computation as *Turing-equivalent* models.

Theorem 6.1 (Turing-equivalent models)**.** *If a computational problem* Π *is solvable in one of the following models of computation, then it is solvable in all of them:*

The Church–Turing Thesis: The above equivalence of many disparate models of computation leads to the Church–Turing Thesis, which has (at least) two different variants:

1.

2.

This is not a precise mathematical claim, and thus cannot be formally proven, but it has stood the test of time very well, even in the face of novel technologies like quantum computers (which have yet to be built in a scalable fashion); every problem that can be solved by a quantum algorithm can also be solved by a RAM program, albeit much more slowly.

Simple and elegant models:

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.

7 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 5.1, 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.

8 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

[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.

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

9 Randomized Algorithms: Definitions

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

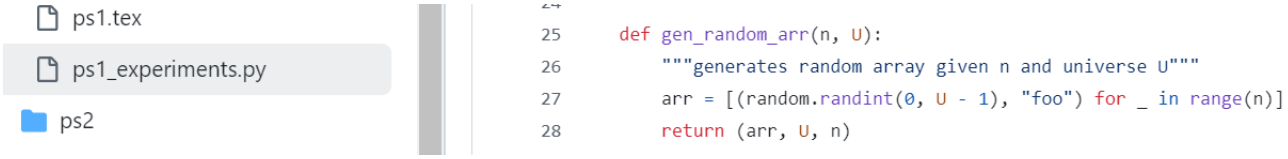


Figure 1: The random.randint command in Python allows you to generate a random integer from $[U]$.

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 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 9.1. 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: