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

Lecture 8: Word-RAM and the (Extended) Church-Turing Thesis

Harvard SEAS - Fall 2025 *2025-09-23*

1 Announcements

PS1 & SRE2 feedback, reflection:

- Median time 10.5hrs, 75th percentile 15hrs.
- How rigorous and detailed should a proof be?
  - Your proofs should always be rigorous, meaning the statements you make should be precise and correct.
  - But the level of *detail* will vary depending on context. Early in the course, we will be expecting more low-level details, to build your skills at verifying your claims. As the course goes on, (y)our proofs will necessarily be taking larger steps. That is, you will still be making a series of precise and rigorous claims, but there will be more details omitted in justifying the claims or how they follow from the previous ones.
  - See the solution sets for examples!
- Questions during lecture.
  - I will try to repeat questions aloud. Remind me if I forget!
  - Remote students, please do ask questions via the Ed thread or in the Zoom chat! One of the staff can then ask the questions out loud.
- Lecture notes and course materials
  - Lecture notes are fully fleshed out in the course textbook (linked from course schedule and on Perusall).
  - I've added more whitespace for your notetaking convenience (whether on paper, on an iPad, or by adding textboxes to the pdf).
- Background materials (e.g. on proofs)
  - See the syllabus, and also https://cs121.boazbarak.org/background/
  - Recommended secondary texts should be also available on reserve through the libraries (I think also with electronic access via Hollis).

- Opportunities to do more SREs on your own: come up with these yourself! Pick out something one of you has studied in the textbook and explain it to a classmate.

My next OH: Thursday 11:15-12 SEC 3.327, Monday 2-2:45pm Zoom.

Harvard Computing Contest Club (HC3) is expanding in scope, to introduce people to competition programming. Talk to Maxwell and fill out interest form: https://forms.gle/HD7qAWeGpuoReEKq8.

Recommended Reading:

- Hesterberg–Vadhan Ch. 7
- MacCormick Ch. 5, 10
- CLRS Section 2.2

2 Loose Ends: The Church–Turing Thesis

As we saw last time, 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 .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:*

- *RAM programs*
- *%-extended RAM programs*
- *Python programs*
- *OCaml programs*
- *C programs (modified to allow a variable/growing pointer size)*
- *Lambda calculus (an early model of computation invented by the mathematician Alonzo* **Church***)*
- *Turing machines (an early model of computation invented by the mathematician Alan* **Turing***)*
- ⋮

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.

3 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. 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. These are used as follows:

- Memory:
- Output:
- Operations:
- Crashing: A Word-RAM program P *crashes* on input x and word length w if any of the following occur:
  1.
  2.
  3.

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). Two sample Word-RAM Programs are given as Algorithms 1 and 2.

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 and using it as our model of computation?

A:

Since Definition .2 allows the same word-RAM program P to be instantiated with different word lengths, 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 define its runtime:

Definition .3. 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.

```
   Duplicate(x):
   Input      : An array x of length n
   Output     : The array x ∘ x, i.e. x concatenated with itself.
   Variables  : input_len, output_ptr, output_len, input_ctr, output_ctr,
                temp, zero, one, three
 0 zero = 0;
 1 one = 1;
 2 three = 3;
 3 input_ctr = 0;
 4 output_ctr = input_len;
 5     temp = input_ctr − input_len;
 6     IF temp == 0 GOTO 13;
 7     temp = M[input_ctr];
 8     MALLOC;
 9     M[output_ctr] = temp;
10     input_ctr = input_ctr + one;
11     output_ctr = output_ctr + one;
12     IF zero == 0 GOTO 5;
13 output_ptr = 0;
14 output_len = three × input_len;
```

Algorithm .1: A Word-RAM program to illustrate the use of MALLOC and the output truncation. At the end of the program, $S = 2n$ but `output_len` $= 3n$, so the output is truncated to end at $M[S-1]$.

3.

Mental model: if $P[w](x)$ crashes, buy a better computer with a larger word size w and try again. With this definition, it can be verified that Algorithms 1 and 2 solve the computational problems they claim to solve.

Definition .4. The *running time* of a word-RAM program P on an input x is defined to be

$$T_P(x) =$$

As we have been doing throughout the course, 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.

Q: What are the running times of Algorithm 1 and 2?

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 .5. *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 aspects of "computational efficiency in practice" not captured by the Word-RAM model?

4 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 .6. *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 Word-RAM is a Turing-equivalent model:

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

Corollary .7. *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 .6 that their runtimes also hold for the Word-RAM model.

Proof Sketch of Theorem .6.

(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 $(B(x)/(\log T(x)))^{O(1)}$.

5 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. (For randomized algorithms, however, it is conjectured that they provide only a polynomial savings.)

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. For this reason, the Extended Church–Turing Thesis is also often called the *Strong* Church–Turing Thesis.

6 Takeaway

The Word-RAM model is the formal model of computation underlying everything we are doing in this book. However, it can be cumbersome to directly work with the intricacies of the Word-RAM model, such as memory allocation and the need for bignum arithmetic. Thus, it is usually more convenient to use the equivalence with a contrained RAM model as given by Theorem .6,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.

7 Turing Machines

Turing Machines can be seen as a variant of the (Word-)RAM model with *constant* word size. Formal definitions are given below in the optional reading, but the main changes are as follows:

1. *Finite Alphabet:*

2. *Memory Pointer:*

3. *Read/write:*

4. *Moving Pointer:*

With these changes, there is a mathematically very elegant description of Turing Machines (which you can find in the textbook), with no arbitrary set of operations being chosen (allowing any "constant-sized" computation to happen in one step). For this reason, Turing machines are the main model covered in most books and classes on the Theory of Computation (such as CS 1210). We work with the Word-RAM Model because it is better-suited for measuring the efficiency of algorithms in practice, and is the main model used (implicitly) in the study of algorithms.

On the surface, Turing Machines seem to be weaker than Word-RAM programs. Nevertheless they are polynomially equivalent:

Theorem .8. *Turing Machines and Word-RAM Programs can simulate each other with only a polynomial slowdown.*

A proof sketch can be found in the textbook.

```
   SameSum(x):
   Input          : An array x of length 4
   Output         : 1 if x[0] + x[1] = x[2] + x[3], 0 otherwise
 0 base = 2;
 1     nextbase = base × 2;
 2     diff = nextbase + 1;
 3     diff = diff − nextbase;
 4     IF diff == 0 GOTO 7;
 5     base = nextbase / 2;
 6     GOTO 1;
 7 FOR i = 0 to 3 DO;
 8     MALLOC; MALLOC;
 9     M[4 + 2 ∗ i] = M[i]%base;
10     M[5 + 2 ∗ i] = M[i]/base;
11 FOR i = 0 to 3 DO;
12 prefixlowsum = M[4] + M[6];
13 prefixhighsum = M[5] + M[7];
14 prefixsumlow = prefixlowsum%base;
15 prefixsumhigh = prefixlowsum/base + prefixhighsum;
16 Similarly calculate suffixsumlow, suffixsumhigh from M[8] + M[10] and
     M[9] + M[11];
17 IF prefixsumlow == suffixsumlow AND
     prefixsumhigh == suffixsumhigh THEN M[0] = 1;
18 ELSE M[0] = 0;
19 output_ptr = 0; output_len = 1;
```

Algorithm .2: A Word-RAM program (with a lot of shorthand for readability) to illustrate the use of bignum and saturation arithmetic. If the program doesn't crash, we know that $x[i] < 2^w$ for each i, but the sum of two elements of $x[i]$ can exceed 2^w. Thus, we calculate $\texttt{base} = 2^{w-1}$ as the largest power of two such that $\texttt{base}+1$ does not saturate at $2^w - 1$, and then decompose each $x[i]$ into its base base representation (with the units digit in $M[4+2i]$ and the $\texttt{base} = 2^{w-1}$ digit in $M[4+2i+1]$). We then calculate the sums by the grade-school addition method, noting that there is no carry past the second digit since the sum of any two input numbers is smaller than $2 \cdot 2^w \leq 2^{2w-2} = \texttt{base}^2$, since $w \geq 3$ due to the presence of the constant 4 in the program. (If we needed to do bignum multiplication, it would be better to have $\texttt{base} = 2^{w/2}$, so that the product of two digits fits always within a word.)

This program uses some operations that are not part of the Word-RAM syntax, but each of them can be decomposed into Word-RAM operations. Since the FOR loop has a constant number of iterations, it can be unrolled. The mod (%) operation can be simulated as in Theorem **??**.