CS 860: Space Bounded Computation **University of Waterloo: Winter 2026**

Lecture 1: Motivating Example and Models

January 6, 2026

Instructor: Sepehr Assadi

Topics of this Lecture

We start our course in this lecture with an example and a review of the main models of computation.

1 Introduction: Space as an Atypical Resource

In traditional algorithm design, *time* is viewed as the central resource, and *space* often plays a secondary role. In this course however, we will do the opposite: our primary concern is understanding the powers and limitations of computation with (very) *limited* space. Space-bounded computation exhibits several atypical and sometimes counterintuitive behaviors that make it fundamentally different from time-bounded computation. To see an example, let us start with a classical result from the 1980s—Barrington's theorem and its simplification by Ben-Or and Cleve—with the following quite informal but flashy message:

"three bits of space" is enough to compute a wide range of functions!

1.1 Motivating Example: Evaluating Circuits with "Three Bits of Space"

Let $n, d \geqslant 1$ and suppose we have a d-depth Boolean circuit $C : \{0,1\}^n \to \{0,1\}$ with $\wedge, \vee, \neg$ gates of fan-in at most two (recall that depth is the length of the longest directed path from any input gate to the output gate). We are additionally given $x \in \{0,1\}^n$ as input to the circuit. Our goal is to evaluate this circuit on the input. See Figure 1 for an illustration.

The question is how much space do we need to do this?

The obvious algorithm. The obvious algorithm for this task is the following recursive algorithm that gets the index of a gate g as input (in the circuit C on x) and evaluate its value.

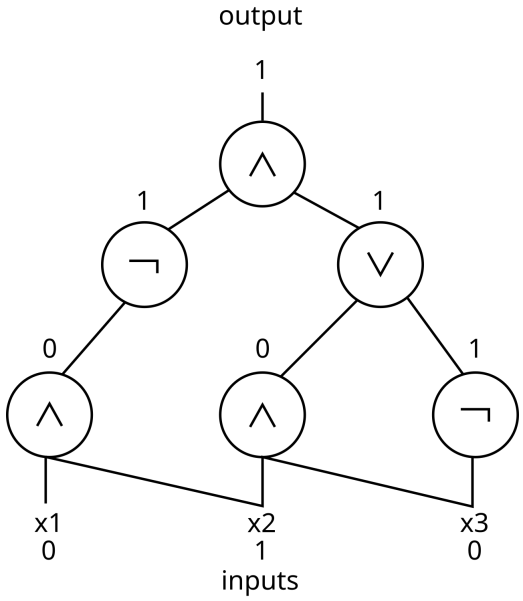

Figure 1: An illustration of a small circuit with input and output gates specified [Use22].

Algorithm 1. `Eval`(gate g):

1. If g is an input gate, say, corresponding to the i-th bit of input, return the value of x_i;
2. If g is a $\neg$ gate with input from gate g', recursively let $b := \texttt{Eval}(g')$ and return $\neg b$;
3. If g is an $\wedge$ gate (resp. an $\vee$ gate) with inputs from gates g' and g'', recursively let $b_1 = \texttt{Eval}(g')$ and $b_2 = \texttt{Eval}(g'')$, and return $b_1 \wedge b_2$ (resp. $b_1 \vee b_2$).

It is easy to see that if we call this algorithm on the output gate, we will recover the correct answer. The question now is how much space do we need? We of course need some space to just follow the instructions of the algorithm, know which gate we are recursively calling the algorithm on, which instruction to go back to after a recursion and so on. But on top of this, and more importantly, when running `Eval` on $\wedge/\vee$ gates, we also need to remember the value of the first recursive call, then reuse the space to recursively compute the value of the second recursive call, and then combine it with the extra bit we stored. Thus, even if we entirely ignore the obvious space needed for "following the code", we additionally need $O(d)$ bits of space for running `Eval`.

But is this $O(d)$ extra space really necessary? We already did a great deal of saving in the space, by *reusing* it during the recursive calls. Specifically, even though running `Eval` on a d-depth circuit involves *two* recursive calls on $(d-1)$-depth circuits, we do not need to pay *twice* the space when going from $(d-1)$-depth to d-depth; this is because we can reuse the same space for one of the recursive calls in the next one also and only need to store one extra bit. This means, the space needed to evaluate a d-depth space is only one more bit than a $(d-1)$-depth circuit (compare this with the runtime of `Eval` instead). However, we can *reuse* the space in an even more clever way as we show next.

The clever algorithm. Let R_1, R_2, R_3 be three 1-bit registers. We are going to solve the circuit evaluation problem by only "carrying" these three registers across our recursive calls regardless of the depth of the circuit. For this, we need to design the following algorithm, which, for lack of a better word, we are going to call `Magic`:

Algorithm 2. `Magic`(gate g, index $i \in [3]$):

1. Do *some magic* such that
$$R_i \leftarrow R_i + value(g)$$
and R_j does not change for any $j \neq i$. Here, $value(g)$ is the value of the gate g in circuit C evaluated on x.

Note: Throughout this algorithm and its proof, we do all the calculations modulo two (so, $1 + 1 = 0$).

Before formalizing the do-some-magic step, let us talk a bit more about the `Magic` algorithm. The goal of this algorithm, say, on the input `Magic`$(g, 1)$, is to start with an initial assignment

$$(\ r_1\ ,\ r_2\ ,\ r_3\ )$$

of the three registers (R_1, R_2, R_3), and finish with the assignment

$$(\ r_1 + value(g)\ ,\ r_2\ ,\ r_3\ ).$$

We should emphasize that we are *not* requiring the algorithm to never change the value of R_2 and R_3 during its execution; we only need it to bring them back to their original value right before it terminates.

If we are able to achieve such a behavior, then to solve the original problem, we will simply start with setting the value of the three registers R_1, R_2, R_3 to be 0, run `Magic`$(g, 1)$ for the output gate g, and return R_1 as the final answer. Our goal is thus to implement the do-some-magic step formally to achieve the required guarantee. We do so for each type of a gate separately. For simplicity, we focus on implementing `Magic`$(g, 1)$ only – `Magic`$(g, 2)$ and `Magic`$(g, 3)$ can be implemented analogously.

- **When g is an input gate x_i**: We simply update $R_1 \leftarrow R_1 + x_i$ and keep R_2, R_3 unchanged.
- **When g is a $\neg$ gate with input from gate g':** We first run `Magic`$(g', 1)$ which updates values of (R_1, R_2, R_3) to be $(r_1 + value(g'), r_2, r_3)$. We then update $R_1 \leftarrow R_1 + 1$ which results in the values to become
$$(\ r_1 + value(g') + 1\ ,\ r_2\ ,\ r_3\ ) = (\ r_1 + (\neg value(g))\ ,\ r_2\ ,\ r_3\ ) = (\ r_1 + value(g)\ ,\ r_2\ ,\ r_3\ ),$$
as desired.
- **When g is an $\wedge$ gate with input from gate g' and g'':** this is effectively the only interesting case of the algorithm. We will do the following steps and keep track of what it does to the original values of the registers (R_1, R_2, R_3):
$$(\ r_1\ ,\ r_2\ ,\ r_3\ ),$$
with the part that changed most recently marked in blue. Also note that our final goal is to bring these values to
$$(\ r_1 + value(g') \cdot value(g'')\ ,\ r_2\ ,\ r_3\ )$$
which corresponds to adding $g' \wedge g''$ to R_1.

  1. Run `Magic`$(g', 2)$:
  $$(\ r_1\ ,\ r_2 + value(g')\ ,\ r_3\ ).$$
  2. Run `Magic`$(g'', 3)$:
  $$(\ r_1\ ,\ r_2 + value(g')\ ,\ r_3 + value(g'')\ ).$$

  (The correctness both steps above are by induction on the depth of the gate called).

3. Update $R_1 \leftarrow R_1 + R_2 \cdot R_3$:

   $$(r_1 + r_2 \cdot r_3 + r_2 \cdot value(g'') + value(g') \cdot r_3 + value(g') \cdot value(g'') \ , \ r_2 + value(g') \ , \ r_3 + value(g'') \ ).$$

   We now need to revert values of R_2, R_3 to r_2, r_3 and get rid of the extra terms in R_1.

4. Run $\texttt{Magic}(g', 2)$:

   $$(r_1 + r_2 \cdot r_3 + r_2 \cdot value(g'') + value(g') \cdot r_3 + value(g') \cdot value(g'') \ , \ r_2 \ , \ r_3 + value(g'') \ )$$

   as it adds $value(g')$ to R_2 holding $r_2 + value(g')$ already, reverting it back to its original value of r_2.

5. Run $\texttt{Magic}(g'', 3)$ again:

   $$(r_1 + r_2 \cdot r_3 + r_2 \cdot value(g'') + value(g') \cdot r_3 + value(g') \cdot value(g'') \ , \ r_2 \ , \ r_3 \ ),$$

   the same exact way as the previous step.

6. Update $R_1 \leftarrow R_1 + R_2 \cdot R_3$:

   $$(r_1 + r_2 \cdot value(g'') + value(g') \cdot r_3 + value(g') \cdot value(g'') \ , \ r_2 \ , \ r_3 \ ),$$

   as it now adds $r_2 \cdot r_3$ to R_1, canceling the same term already in R_1.

7. Run $\texttt{Magic}(g', 2)$:

   $$(r_1 + r_2 \cdot value(g'') + value(g') \cdot r_3 + value(g') \cdot value(g'') \ , \ r_2 + value(g') \ , \ r_3 \ ).$$

8. Update $R_1 \leftarrow R_1 + R_2 \cdot R_3$:

   $$(r_1 + r_2 \cdot value(g'') + value(g') \cdot value(g'') + r_2 \cdot r_3 \ , \ r_2 + value(g') \ , \ r_3 \ ).$$

9. Run $\texttt{Magic}(g', 2)$:

   $$(r_1 + r_2 \cdot value(g'') + value(g') \cdot value(g'') + r_2 \cdot r_3 \ , \ r_2 \ , \ r_3 \ ).$$

10. Run $\texttt{Magic}(g'', 3)$:

    $$(r_1 + r_2 \cdot value(g'') + value(g') \cdot value(g'') + r_2 \cdot r_3 \ , \ r_2 \ , \ r_3 + value(g'') \ ).$$

11. Run $R_1 \leftarrow R_1 + R_2 \cdot R_3$:

    $$(r_1 + value(g') \cdot value(g'') \ , \ r_2 \ , \ r_3 + value(g'') \ ).$$

12. Run $\texttt{Magic}(g'', 3)$:

    $$(r_1 + value(g') \cdot value(g'') \ , \ r_2 \ , \ r_3 \ ),$$

    which is exactly the state we wanted to get to.

13. **When g is an $\vee$ gate with input from gate g' and g'':** We can go through a similar step as before, our simply use the De Morgan's that $a \vee b = \neg \, (() \, \neg a \wedge \neg b)$, which can be handled by a direct reduction to the previous two cases.

This concludes the description $\texttt{Magic}$.

All in all, we saw that using $O(1)$ recursive calls and another $O(1)$ instructions[1] at each level, we can evaluate a depth-d gate using only three one-bit registers, much more space efficiently than the direct way

[1] We should emphasize that the above implementation is not the most efficient one in terms of constants and we could have been more careful in ordering the instructions to reduce their numbers; however, we went with the most straightforward way to shed more light into the already magical looking approach of $\texttt{Magic}$.

of `Eval`. We do note this does not mean `Magic` can be implemented on your computer using just 3 bits of space, since we ignored the space needed for "following the code" (which we purposefully leave vague and not formalized at this stage), but morally speaking, we can see `Magic` as a way more space-efficient variant of `Eval`. This showcase a peculiar nature of space that allows for a very high degree of *reusability* compared to time and most other resources of interest.

We now start formalizing our study of space in this course by defining the computational models of interest for us. We will be studying both *uniform* models as well as *non-uniform* ones. Roughly speaking, in uniform models, such as Turing machines or RAM machines, a single algorithm describes the computation for all input lengths. In contrast, non-uniform models, such as Boolean circuits or branching programs, allow a separate computational object for each input length, with no requirement that these objects be generated efficiently (thus, among other things, the non-uniform models also allow us to distinguish between the space needed for "following the code" vs "actual computation").

2 Non-uniform Model: Branching Programs

The main non-uniform model we study in this course is **branching programs**:

Definition 1. For integers $w, d, n \geqslant 1$, a **branching program** (BP) with **width** w and **depth** d on n-bit inputs is a layered directed acyclic graph (DAG) with $d+1$ layers satisfying the following properties:

- There are $d+1$ layers of vertices, with a single vertex in the first layer, two vertices in the last layer, and w vertices in every layer in the middle.
- Every vertex v except for the last layer is labeled by a number $i_v \in [n]$. The two vertices in the layer are labeled 'accept' and 'reject'.
- Every vertex except for the last layer has two outgoing edges labeled 0 and 1.

The computation of the branching program on an input $x = (x_1, \ldots, x_n) \in \{0,1\}^n$ proceeds as follows:

Starting from the start vertex in layer 0, in each step the program reads the label i_v of the current vertex, let $b := x_{i_v}$, and take the edge labeled b out of v to the next layer. Once the program reaches the final state, it accepts or rejects the input depending on the label of the reached vertex.

A branching program thus allows us to compute a function $f : \{0,1\}^n \to \{0,1\}$ for a *fixed* choice of n (hence the term non-uniform, meaning that a BP only needs to work for a specific length of inputs).

Let us see a couple of examples a straightforward branching programs for computing any function f. Before that, the following remark is in order.

Remark. Roughly speaking, a branching program of width w and depth d can simulate a "standard" algorithm that uses $\log w$ bits of space and d time. We will formalize this in the next section once we defined what a "standard" algorithm means, but an intuitive way to see this is to think of each layer of the BP as capturing a time step of the algorithm, and each vertex in a layer, as a possible combination of all the bits in the memory of the algorithm.

Trivial branching programs. There are two immediate ways of designing a branching program for a function $f : \{0,1\}^n \to \{0,1\}$, one of them is extremely width-inefficient and depth-efficient, and the other is the exact opposite.

Proposition 2. *For any $n \geqslant 1$, any function $f : \{0,1\}^n \to \{0,1\}$ admits both of the following:*

1. *a branching program with width 2^n and depth $O(n)$;*
2. *a branching program with width $O(1)$ and depth $2^{O(n)}$.*

Proof. We prove each part separately:

1. The idea is to read the input once, branching on each bit, and explicitly encode the entire input string in the state.

   Construct the following BP with $n+1$ layers: The first layer is indexed by 0 and has one vertex. For each layer $i = 1$ to $n-1$, layer i contains 2^i vertices named by a distinct string $y \in \{0,1\}^i$ and has label $i+1$. The two edges going out of vertex $y \in \{0,1\}^i$ in layer i go to vertices $y0, y1 \in \{0,1\}^{i+1}$ based on the value of x_{i+1}.

   Thus after reading all input bits, the computation reaches a unique vertex in the second to last layer named the same as the input $x \in \{0,1\}^n$. Finally, from each vertex in that layer named $y \in \{0,1\}^n$, add an edge to the accept state if $f(y) = 1$ and to the reject state otherwise.

   It is easy to see that this is a BP with 2^n width and $n+1$ depth that computes f correctly.

2. We do the opposite in this case: enumerate all possible inputs one by one, and check whether the actual input matches the current candidate.

   Fix an arbitrary ordering of all strings $y \in \{0,1\}^n$. For each string y, we construct an $O(n)$-depth subprogram that checks whether the input $x = y$ or not. If it is, the entire BP moves to an accept or reject state depending on the value of $f(y)$ and continue to do so until it reaches the last layer and solve the problem. Otherwise, it moves to the next subprogram for the next string y. Each subprogram for y scans the bits $x_1, \ldots, x_n$ sequentially and as long as x is consistent with y it goes to a next 'continue' state in the next layer, otherwise it goes to a 'fail' state. The continue states do the same until they scan all n bits of y against x and at the end, they either go to an 'accept' state or a 'reject' one based on the value of $f(y)$ and those states continue to the same type of state in the next layer throughout the rest of BP until they reach a final accept or reject state. A 'fail' state instead simply goes to the next 'failed' state in the next layer and continue like that until the subprogram finishes and then go to first state of the subsequent subprogram.

   It is easy to see that this is a BP with $O(1)$ width and $O(n \cdot 2^n)$ depth that computes f correctly. With some care, the width of this BP can be reduced to 3; one can also prove that width 2 BPs cannot compute certain functions (say $f(x) = 1$ iff $\|x\|_0$ is divisible by three) regardless of their depth. In this course, we will not be concerned with specific constants in width and depth of BPs and only consider them asymptotically.

□

Proposition 2 suggests that when measuring space and time in the language of branching programs (and not their actual definition), every function $f : \{0,1\}^n \to \{0,1\}$ can be computed either in linear space and linear time, or constant space and exponential time. Of course this is not the case when looking at uniform models of computations. However, we will be using branching programs primarily for proving *lower bounds* and those lower bounds also can be carried out to almost any other reasonable model of uniform computation (we will discuss this further as well later).

Remark. A corollary of Proposition 2 is that BPs—in this formulation—can only really capture the notion of space below linear space, and time, below exponential time, in the sense that, we can never prove stronger lower bounds. Our focus throughout this entire course will be primarily limited to

logarithmic and sublinear-space and polynomial time algorithms and thus we can, in principle, use BPs to prove lower bounds for such computation.

Let us now use our ideas from the motivating example to show constant-width BPs can solve very non-trivial problems even in much lower depth than the bounds of Proposition 2.

2.1 Power of Constant-Width Programs: Barrington's Theorem

The following classical theorem due to Barrington proves a striking power for constant-width branching programs (and thus "constant space" non-uniform algorithms).

Theorem 3 (Barrington's Theorem [Bar89])**.** *Any Boolean circuit $C : \{0,1\}^n \to \{0,1\}$ with $\wedge, \vee, \neg$ gates of fan-in at most two and depth d can be simulated by a branching program of width 5 and $O(4^d)$ depth.*

We will prove a simplification of this result due to Ben-Or and Cleve [BC92] that gets slightly weaker bounds quantitatively but qualitatively is more general and is also easier to describe. In particular, we show a branching program with width 8 and $2^{O(d)}$ depth for some unspecified constant in the exponent.

A careful reader may realize that we have actually already proved this result! The magical "three bits of space" algorithm we discussed earlier is precisely the proof for this result. At a high level, we can create a branching program with width 8, corresponding to tuples $(r_1, r_2, r_3) \in \{0,1\}^3$, as assignments to the three registers (R_1, R_2, R_3) and then follow the `Magic` algorithm. We sketch the ideas below.

For a gate g in the circuit C and index $i \in [3]$, let us call $B(g, i)$ a (slightly modified) branching program that on every layer has 8 vertices and has no start or accept or reject state. Instead, starting from any state (r_1, r_2, r_3) in the first layer, if we follow this branching program, we should end up in the state (r'_1, r'_2, r'_3) where $r'_i = r_i + value(g)$ and $r'_j = r_j$ for $j \neq i$. Note that in `Magic`, the two operations used by the algorithm, beside recursion, were

$$R_i \leftarrow R_i + c \qquad \text{and} \qquad R_i \leftarrow R_i + R_j \cdot R_k,$$

for some fixed $c \in \{0,1\}$ that can come either from a recursion or from reading the input. We can implement either of these operations in a branching program using a single layer, by connecting the states according to the permutation these operations induce (in the former case, this is the step that involves reading something from the input). Finally, a BP for a gate g can be obtained by concatenating BPs for gates g' and g'' interleaved with the operations above exactly as dictated by `Magic`. The width of this BP remains 8 through the concatenation, and its length involves $O(1)$ copies of the BPs for one lower depth gates. Thus, the depth increases exponentially with d leading to $2^{O(d)}$ depth at the end.

This concludes the proof sketch.

Remark. The class of all problems computable by polynomial-size circuits of depth $O(\log n)$ is referred to as NC^1. Theorem 3 implies that NC^1 can be simulated with poly-size constant-width branching programs. This is one of the earliest examples showing that space-bounded non-uniform computation can capture sophisticated complexity classes.

3 Uniform Model: Random Access Memory (RAM) Machines

While branching programs are convenient for non-uniform computation, most algorithm design is done in uniform models such as Turing machines or RAM machines. Since the RAM model is the one we will use for describing algorithms in this course, we review it here more carefully.

Intuitively, a RAM machine is meant to capture the way real computers operate: they have a small number of registers and a large memory array that can be indexed arbitrarily. The formal definition is

as follows. For this definition (and the rest of this lecture), we follow the upcoming textbook of [Vio25], although we modify the definition and restrict it further to make it suite our specific focus on space bounded computation (with not "too much space", namely, restricted to polynomial space algorithms).

Definition 4. A **Random Access Memory (RAM) machine** with parameters $b, k \in \mathbb{N}$ consists of:

- A read-only array $I[1], \ldots, I[n]$ of input bits, where n can be any arbitrary integer (the machine works for all choices of $n \in \mathbb{N}$).
- k registers $R[1], \ldots, R[k]$, each capable of holding a w-bit string for $w = b \cdot \log n$.
- A read-write array $M[1], M[2], \ldots$ of memory bits. We do not limit the length of the memory apriori and instead the algorithm can decide how much of the memory it is going to use (although as it will become clear, it can index to at most n^b entries of this memory anyway).
- A set of k instructions that include register assignments, arithmetic calculations, reads from input, writes to memory, conditional GOTO statements, and ACCEPT and REJECT statements. We will specify these instructions after this definition.
- An instruction counter *cnt* pointing to the next instruction to execute.

The computation of the machine on an input x starts by assuming x is placed in I which defines n also, M and R are initialized to all 0, and follows the instructions one by one using the counter *cnt*, jumping between the instructions using the GOTO statements, until it reaches an ACCEPT or REJECT statement.

The **runtime** of a machine on a specific input x is the total number of instructions run throughout the computation, and its **space** is the largest index of M that was ever accessed (either a read or a write).

The crucial point is that k and b in the machine are absolute constants, and the machine has a fixed number of registers, and the word-size of those registers can grow with $\log n$ based on the given input size.

To further formalize the notion of computation in a RAM machine (on a fixed input x), we define a **configuration** which is a 3-tuple

$$(cnt, R, M),$$

that records the value of *cnt* as well as registers R and memory array M at any point of the computation. The original configuration of the machine is

$$(1, (0^w)^k, 0^*).$$

Running through the instruction of the machine then updates these configurations to $C_1, C_2, \ldots, C_t$ where C_t is either $(ACCEPT, R, M)$ or $(REJECT, R, M)$ for some assignment of R and M. This way, t will be runtime of the machine on input x. It only remains to specify how the instructions update these configurations, and for that we need to formalize the instruction-set itself also.

Instruction set in RAM machines. We allow the following instructions in a RAM machine and define their meaning as the way they update a configuration:

- **Register assignment:** $R[i] \leftarrow X$ where X can be any of the following

  $$I[R[j]] \quad , \quad M[R[j]] \quad , \quad R[R[j]], \quad y \in \{0,1\}^w .$$

  The meaning of these operations is to place the value index by $R[j]$ inside any of the arrays I, M or R, or a fixed value $y \in \{0,1\}^w$ inside the register $R[i]$ (in the case of I and M that are assigning a bit to a word, we interpret them as updating the least significant bit of $R[i]$ and setting the rest to be 0).

After running an instruction of this type, say, $R[i] \leftarrow R[R[j]]$, the configuration is updated as:

$$(cnt, R, M) \to (cnt + 1, R', M),$$

where $R'[\ell] = R[\ell]$ for all $\ell \neq i$ and $R'[i] = R[R[j]]$ if $R[j] \in [k]$ and 0 if $R[j] \notin [k]$.

- **Memory write:** $M[R[i]] \leftarrow R[j]$. After running an instruction of this type, the configuration is updates as:

  $$(cnt, R, M) \to (cnt + 1, R, M'),$$

  where $M'[\ell] = M[\ell]$ for all $\ell \neq R[i]$ and $M'[R[i]] = least(R[j])$, where $least(\cdot)$ returns the least significant bit of its input.

- **Arithmetic calculation:** $R[i] \leftarrow E$ where E is a binary (or unary) expression involving two registers $R[j]$ and $R[\ell]$ in the RHS and one of the operators $+, -, \times, \div, \%$ (addition, subtraction, multiplication, (integer) division, and modular remainder), as well as bit-wise $\wedge, \vee, \neg$. The operations are implemented over $\mathbb{N}$ and only store the least w words of the result in $R[i]$. After running an instruction of this type, the configuration is updated in a natural way as above.

- **GOTO instruction:** GOTO i IF E where $i \in [k]$ and E is a Boolean predicate involving one or two registers $R[j]$ and $R[\ell]$ and one of the operators $>, \geqslant, <, \leqslant, =$ between them. This instruction updates the configuration as

  $$(cnt, R, M) \to (cnt', R, M),$$

  where $cnt' = cnt + 1$ if E is false and $cnt' = i$ if E is true.

- **Terminate instruction:** ACCEPT , REJECT: this is the final instruction of a computation which decides if the input is accepted or rejected, respectively.

This concludes the description of a RAM machine, the uniform model of computation we work with in this course.

3.1 Complexity Classes

Using RAM machines, we can now introduce our usual complexity classes.

Whenever there exists functions $t : \mathbb{N} \to \mathbb{N}$ and $s : \mathbb{N} \to \mathbb{N}$ such that on any input $x \in \{0,1\}^*$, the runtime and space of P is upper bounded by, respectively, $t(length(x))$ and $s(length(x))$, we say that P has **time complexity** $t(\cdot)$ and **space complexity** $s(\cdot)$ (here, $length(x)$ is the number of bits in x, namely, the parameter n of the input size).

Our usual complexity classes are as follows:

- $\mathsf{Time}(t(\cdot))$: the set of all languages $L \subseteq \{0,1\}^*$ such that there exists a RAM machine with time complexity $t(\cdot)$ that accepts precisely every $x \in L$;
- $\mathsf{Space}(s(\cdot))$: the set of all languages $L \subseteq \{0,1\}^*$ such that there exists a RAM machine with space complexity $s(\cdot)$ that accepts precisely every $x \in L$;
- $\mathsf{TiSp}(t(\cdot), s(\cdot))$: the set of all languages $L \subseteq \{0,1\}^*$ such that there exists a RAM machine with time complexity $t(\cdot)$ *and* space complexity $s(\cdot)$ that accepts precisely every $x \in L$.

Remark. There is no reason for $\mathsf{TiSp}(t, s) = \mathsf{Time}(t) \cap \mathsf{Space}(s)$ since in the TiSp definition, we require a *single* machine to have both bounded time and space, whereas for the RHS, it is possible for one machine to have a bounded time, and another to have bounded space.

The familiar complexity classes that we will be working most in this class are:

- Polynomial time:
$$\mathsf{P} := \bigcup_{a \in \mathbb{N}} \mathsf{Time}(n^a),$$
where by 'n^a' we mean the function $t : \mathbb{N} \to \mathbb{N}$ such that $t(n) = n^a$; these are the problems that can be solved in polynomial time.

- Logarithmic space:
$$\mathsf{L} := \mathsf{Space}(0)$$
which means we do not use any extra memory and the entire computation is done using the registers in the program.[2].

- Steve's class:
$$\mathsf{SC} := \bigcup_{a,b \in \mathbb{N}} \mathsf{TiSp}(n^a, (\log n)^b);$$
these are the problems that can be solved simultaneously in polynomial time and polylogarithmic space.

This concludes our first lecture.

References

[Bar89] David A. Mix Barrington. Bounded-width polynomial-size branching programs recognize exactly those languages in nc[1]. *J. Comput. Syst. Sci.*, 38(1):150–164, 1989. 7

[BC92] Michael Ben-Or and Richard Cleve. Computing algebraic formulas using a constant number of registers. *SIAM J. Comput.*, 21(1):54–58, 1992. 7

[Use22] User:Peppeg. Example boolean circuit. https://commons.wikimedia.org/wiki/File:Three_input_boolean_circuit.svg, 2022. Wikimedia Commons, licensed under CC BY-SA 4.0. 2

[Vio25] Emanuel Viola. Mathematics of the impossible: The complexity of computation. *2023-2025 Emanuel Viola, to be published by Cambridge University Press*, 2025. 8

[2]That are constantly many and thus overall use $O(\log n)$ "space" under the notions of space in say, Turing Machines, unlike our word RAM model.