Computational complexity studies the amount of resources necessary to perform given computations. The most relevant resource is computation time – the number of steps necessary to come up with the correct answer. We defined the running time of a Turing Machine, argued that this notion roughly captures the time complexity of algorithms, and saw that more time allows Turing Machines to decide more problems. We then looked at circuits, a non-uniform version of Turing Machines, and showed that lower bounds on circuit size give lower bounds on running time.

Our strategy to understand the limitations of efficient computation was to restict the computational model as much as we need until we could prove something about its limitations. That's how we ended up studying AC^0 circuits. But AC^0 circuits turned out to be too unrealistic to capture even simple computations like XOR. Unfortunately, once we give our circuits more power – say by going from AC^0 to NC^1 – we don't know how to prove which things are hard for them anymore.

Today we will look at another attempt to refine our understanding of efficient computation. Not all time-efficient computations are made equal. For example consider these two programs that reverse an array of n integers:

```
int [] r1( int [] x) {
  int n = length(x);
  int [] y = new int[n];
  for (int i = 0; i < n; i++) y[i] = x[n-i-1];
  return y;
}
```

and

```
int [] r2( int [] x) {
  int n = length(x), tmp;
  for (int i = 0; i <= n/2; i++) {
    tmp = x[n-i-1];
    x[n-i-1] = x[i];
    x[i] = tmp;
  }
  return x;
}
```

Both of these programs reverse the array `x` in linear time, but there is a difference: To do the reversal, the first program needs to allocate an additional array `y` of n integers, while the second one can do the reversal in place. So program `r2` is not only time-efficient, but also memory-efficient. Today we begin to study how the use of memory, or *space*, affects efficient computation.

1 Space-bounded Turing Machines

When we define the computation space of a Turing Machine, we have to be a bit more careful than when we talk about time. The reason is that while a Turing Machine needs at least linear time to do most things (although there are exceptions), a lot of interesting tasks can already be done in sublinear space. For example suppose we want to compute the majority of n bits. You can convince yourself that this takes must take at least linear time (because all input bits must be read), but it can be done in space $O(\log n)$ – at each step it is sufficient to keep track of the sum of the bits seen so far.

To account for cases like this one, when we talk about space-bounded Turing Machines, we will have the following model in mind. Let's start by describing Turing Machines for decision problems. The machine has two tapes, a read-only *input tape* and a read-write *work tape*. The transitions of the TM can depend on the contents of the current cell on both tapes, but we are only allowed to write symbols on the work tape. The computation ends when the TM writes an answer (0 for reject, 1 for accept) on the work tape and it goes into a special halting state. (We will also assume (without loss of generality) that before the TM halts both tape heads are returned to the initial position.) The *space complexity* of TM M on input x is the total number of cells ever used (i.e. the rightmost position of the head) on the work tape.

When the TM is required to write an answer (for example, it is solving a search problem) it has an additional write-only, one-way *output tape* where the answer will be written. The head on the write tape can only stay in place or move to the right.

We say a Turing Machine M runs in space $s(n)$ if for all $x \in \{0,1\}^n$, the space complexity of M on input x is at most $s(|x|)$.

In the case of non-deterministic Turing Machines, we will require that the non-determinism tape is not only read-only but also one-way – the head always moves left to right. This requirements models the intuition that when the space of our computation is bounded, we should not even have enough space to remember our previous non-deterministic choices, so once a non-deterministic choice is "used", it is forgotten (unless the machine stores it explicitly on its work tape).

We say a non-deterministic Turing Machine N runs in space $s(n)$ if for every input x and all non-deterministic choices N is allowed to make, the space complexity of N on input x is at most $s(|x|)$.

Clearly a TM that runs in time $t(n)$ also runs in space $t(n)$. Therefore to refine our understanding of polynomial-time Turing Machines we need to impose a time bound which is less than polynomial in n. The simplest scenario we may wish to model are computations that run "in-place", like the program `r2` above. How much of the work tape do such computations use? Initially you may think that an in-place computation use only constant memory, as all they need are some local counters like `n` and `tmp`. But if you think carefully you realize that these counters in fact use $\log n$ cells of memory: The input array `x` has length n, so these counters that eventually need to index all positions in `x` will take up $\log n$ bits each.

So intuitively, even if we want to make a single pass over an input of length n, we need at least $\log n$

cells on the work tape.[1] Therefore it makes sense to look at *logarithmic space* Turing Machines as the simplest realistic, space-bounded model of efficient computation.

Definition 1. The class L consists of all decision problems that can be decided on a Turing Machine that runs in space $O(\log n)$ on inputs of length n. The class NL consists of all decision problems that can be decided on a nondeterministic Turing Machine that runs in space $O(\log n)$ on inputs of length n.

Notice that our definition does not make any reference to time complexity – can an algorithm that runs in logarithmic space take exponential time? In fact not: If M runs in logarithmic space, then M must run in polynomial time. We reason as follows. A computation on M on input x can be viewed as a walk on the *configuration graph* of M on input x. This is a directed graph whose nodes are triples (q, i, w), where q ranges over all states of M, i is an integer that marks the position of the head on the input tape, and w ranges over all possible contents of the work tape (including a special marker for the head). The graph contains an edge from (q, i, w) to (q', i', w') if this represents a valid transition in a computation of M on input x. If M runs in logarithmic space, when x has length n, the number of nodes in the graph is $O(1) \times n \times 2^{O(\log n)} = n^{O(1)}$. Now notice that in a halting computation on M on x, no node in the graph can be repeated twice, so the computation must terminate within a polynomial number of steps.

In particular, this argument shows that L $\subseteq$ P. Could it be the case that L = P? We do not know, but most computer scientists believe this is not the case. The conjecture L $\neq$ P formalizes the belief that certain efficient computational tasks are *required* to use additional memory. In this context, it can be shown that certain tasks like linear programming or maximum flow (formalized appropriately) are P-complete (under log-space reductions) in the sense that if any of these tasks could be performed in logspace, then L would equal P. This explains why we don't have in-place algorithms for these tasks.

Some of the simplest efficient algorithms hat use external memory are *depth first search* and *breadth first search*. These algorithms solve the decision problem graph connectivity:

CONNECTIVITY: Given an graph, is it connected?

Is the use of external memory necessary when we solve connectivity? As we will see soon, in the case of directed graphs, this question essentially asks if L is different from NL. What about undirected graphs? In 2005 Omer Reingold showed that undirected graph connectivity can be solved in logarithmic space! So external memory is not necessary to tell if a graph is connected or not. Since L $\subseteq$ P, Reingold's algorithm runs in polynomial time – but the running time could be a very large polynomial in the graph size.

Before we turn to these fascinating questions, let us take a detour into non-uniform models of small space computations. Just like circuits helped us say something about the limitations of polynomial-time Turing Machines, we now look into *branching programs* – the "hardware" model of small space algorithms.

[1] Actually, while this should be clear for Random Access Machines, it is not necessarily true for Turing Machines, because a TM can make a pass over its input left-to-right without ever storing any index. But even if we want to make two independent passes (i.e. keep two counters) on a TM with $o(\log n)$ space we run into trouble.

2 Branching programs

A branching program is a device with some small number of states. Before it starts its computation, the device decides how it is going to process its input: Maybe first it looks at the input bit x_5, then x_2, x_7, x_2 again, and so on. At each time step, it updates its state as a function of its current state and the input bit it was looking at. However, the device can use different update rules at different points in time.

Definition 2. A *branching program* on n inputs of width w and length t consists of a sequence of input positions $k_0, \ldots, k_{t-1} \in \{1, \ldots, n\}$ and functions $f_0, \ldots, f_{t-1} \colon \{0, \ldots, w-1\} \times \{0,1\} \to \{0, \ldots, w-1\}$.

On input $x_1, \ldots, x_n$, the branching program computes as follows. Initially, the program is in state $s_0 = 0$. In time step i, the state of the branching program is updated via the rule $s_{i+1} = f_i(s_i, x_{k_i})$. At time t, the branching program accepts if it is in state 0, and rejects if it is in any of the other states.

When $w = 2^k$ we can think of the branching program as this. The program has a fixed set of registers $R_1, \ldots, R_k$, initialized to 0, and a program consisting of t "instructions" $g_0, \ldots, g_{t-1}$. Each instruction is of the type "look up some input bit and update the registers". In the end the program accepts (say) if $R_1 = \cdots = R_k = 0$.

Theorem 3. *If $L \in \mathrm{L}$, then L can be computed by a family of branching programs of polynomial length and polynomial width.*

Proof sketch. Let M be a logarithimic space Turing Machine for L. We will first argue that we can make the motion of the head of M's input length completely independent of the input (and only dependent on the input length). The idea is to reprogram the Turing Machine so that its input tape head always makes a move to the right at each time step, until it reaches the end of the tape, and then comes back straight to the beginning. To do this, anytime the head wants to move left, we make a copy of the head position on the work tape, and keep moving the input head to the right until it reaches the end, go back to the beginning, and move it to the left until right before the correct place is reached. We omit the technical details.

Now think of the $k = O(\log n)$ cells of the tape as registers in the branching program. Then each step of M specifies how the register contents are updated as a function of the input bit x_{k_i} read at time i. This process is modeled by a branching program of width $2^{O(\log n)} = n^{O(1)}$. Since every logarithmic space TM runs in polynomial time, the branching program will also have length polynomial in n. □

What about branching program families of smaller width? A branching program whose length is exponential in n can compute all functions in n variables, even if its width is 3. But if we restrict the length to be polynomial in n, it is not clear how much the branching programs can do.

How do branching programs compare with circuits? Polynomial-length branching programs with a constant number of states can be simulated in NC^1. For a branching program B of length t and

width w, we show how to build a fanin 2 circuit of depth $O(\log w \log t)$. In fact this circuit will compute a function $C(s,t,x)$ where s and t are states of B, $x \in \{0,1\}^n$ is an input, and

$$C(s,t,x) = \begin{cases} 1, & \text{if on input } x,\ B \text{ goes from state } s \text{ to state } t \\ 0, & \text{otherwise.} \end{cases}$$

To construct C, we split B in two parts B_1 and B_2 of equal length. Suppose we have already constructed circuits C_1 and C_2 for them. Then we write

$$C(s,t,x) = \bigvee_{u=1}^{w} C_1(s,u,x) \wedge C_2(u,t,x)$$

which describes the fact that if on input x, B goes from state s to state t, then it must do so thru some state w in the middle. The depth of C is then $O(\log w)$ bigger than the depths of C_1 and C_2. Since C_1 and C_2 describe branching programs of half the length, we can continue the construction recursively and obtain a circuit of depth $O((\log w)(\log t))$ for B. (In the base case $t = 1$, C depends on only one bit of x so it can be computed by a circuit of depth $O(\log w)$ also.)

So constant width branching programs can be simulated in NC^1. Can we come up with examples of functions that are hard for them? The XOR function, which was hard for AC^0, can be computed by width 2 branching programs of length n. What about majority?

3 Barrington's theorem

In the 1980s it was widely conjectured that constant-width, polynomial size branching programs cannot compute the majority function. People also believed that the computational power of branching programs should increase as the width gets larger: For instance, polynomial-size branching program families of width 20 are more powerful than those of width 10. In a surprising turn in 1986, David Barrington showed that these beliefs are wrong!

Theorem 4. *If f has a depth d formula, then it has a branching program of width 5 and size $2^{O(d)}$.*

By this theorem logarithmic depth fan-in 2 circuit families, polynomial-size formulas, and polynomial-size branching program families of any constant width above 5 are all equivalent models of computation.

We will prove Barrington's theorem but with the constant 5 replaced by 8. Recall that a branching program of width 8 can be viewed as a machine with 3 registers taking values in $\{0,1\}$. Let's call the register contents of these 3 registers A, B, and C.

Assume f has a formula of depth d. We begin by changing the $\vee$ and $\wedge$ gates in the formula into $\times$ and $+$ gates. These gates compute multiplication and addition over the binary field $\mathbb{F}_2$, respectively. We can perform the gate replacement using the following rules:

$$x \wedge y = x \times y \qquad x \vee y = 1 + (1 + x) \times (1 + y).$$

We obtain a formula for f with $\times$ and $\oplus$ gates and depth $O(d)$. This formula has some extra leaves that are labeled by the constant 1.

We now design a branching program for the formula f. We will prove the following statement by induction on the depth d' of f: There is a branching program of width 8 and size $4^{d'}$ so that when the branching program starts with register contents A, B, and C, it ends its computation with register contents A, B, and $C + f(x_1, \ldots, x_n)B$. The theorem then follows by initializing the registers to $A = 0$, $B = 1$, and $C = 0$.

We prove the inductive statement by looking at the top gate of f. If this gate is the constant 1 or a literal x_i or $\overline{x}_i$, then there f can be computed by a branching program of length 1. If $f = f_1 + f_2$, then we obtain a linear length branching program for g by combining the programs P_1 for f_1 and P_2 for f_2 like this:

$$(A, B, C) \xrightarrow{P_1} (A, B, C + f_1 B) \xrightarrow{P_2} (A, B, C + (f_1 + f_2)B)$$

By inductive hypothesis, each of P_1 and P_2 has length $4^{d'-1}$, so f has length $2 \cdot 4^{d-1} \leq 4^d$. If $f = f_1 \times f_2$, we combine P_1 and P_2 again as follows:

$$\begin{aligned}(A, B, C) \xrightarrow{P_1} (A + f_1 B, B, C) \xrightarrow{P_2} (A + f_1 B, B, C + f_2(A + f_1 B)) \\ \xrightarrow{P_1} (A, B, C + f_2 A + f_1 f_2 B) \xrightarrow{P_2} (A, B, C + f_1 f_2 B).\end{aligned}$$

In each of the steps, the program P_1 or P_2 is applied but the registers are permuted in some order. Using the inductive hypothesis, f has length $4 \cdot 4^{d'-1} = 4^{d'}$, concluding the inductive argument.

4 Nondeterministic logarithmic space

In contrast to deterministic polynomial-time computations, nondeterministic polynomial-time computations are not believed to be realistic. How much power does nondeterminism add to logarithmic space? We will see that every nondeterministic logarithmic space TM can be simulated in deterministic polynomial time, so NL in fact describes a realistic model of computation.

What about the relationship between NL and L? While there appears to be overwhelming evidence that P $\neq$ NP, the "answer" to the L versus NL question seems much less clear. We will see that many of the difficulties we encounter in simulating nondeterministic time efficiently go away when we are dealing with nondeterministic space. Despite this we do not know how to eliminate nondeterminism altogether from space-efficient computations.

Just like SAT embodies the difficulty of NP, the following problem plays the same role for NL:

PATH: Given an directed graph G and two vertices s and t in G, is there a path from s to t in G?

Sometimes it is convenient to work with the following version:

CONN: Given an directed graph G, is it strongly connected (i.e. is there a directed path from every node to every other node)?

We say a decision problem A *logspace reduces* to decision problem B if there is a logarithimic space TM that maps instances x for A into instances y for B so that $x \in A$ if and only if $y \in B$. We say L is NL-complete (under logspace reductions) if every A in NL reduces to L.

Theorem 5. PATH *and* CONN *are* NL*-complete.*

Since graph connectivity can be checked in polynomial time, and logspace reductions run in polynomial time, it immediately follows that NL $\subseteq$ P.

Proof. To see that PATH is in NL we need to give a nondeterministic logarithmic space algorithm that, on input a graph G, checks that it is possible to get from vertex s to vertex t. The nondeterministic tape describes a path from x to y in G, that is a sequence of vertices $v_0, v_1, \ldots, v_k$ where $v_0 = s$, $v_k = t$, and v_i connects to v_{i+1}. The following NTM then checks that the graph is connected by going over all such vertices and checking they are all valid:

On input a graph G and vertices s and t:
    Guess a vertex v (i.e., read it from the nondeterminism tape).
    If $v \neq s$, reject.
    For $i := 1$ to (at most) n:
        Guess a vertex v'.
        If (v, v') is not an edge in G, reject.
        If $v' = t$, accept.
        Otherwise, set $v := v'$.
    If i reached the value n, reject.

This NTM uses three counters: v, w, i, each of which described a number between 1 and n. So it can be implemented in logspace. We can get a logarithmic space NTM for CONN by running the same algorithm for all pairs of vertices (s, t).

We now argue that every L in NL reduces to PATH. Let N be a logarithmic space NTM for L. Given an input x for N we need to construct a triple (G, s, t) so that N accepts x if and only if there is a path from s to t in G. The vertices of G will be the configurations of N, and there is an edge (x, y) if the transition from x to y can be performed in a single computation step. Remember that this can be decided by looking at the "3×2 window" centered around the work tape of x and the corresponding work tape of y and checking that it describes a valid change of configuration (for some choice of nondeterminism). We also need to check that the head position of the input tape is updated accordingly. All of these checks can be done in logarithmic space. We set s to be the vertex that describes the starting configuration of N, and t describes the accepting configuration of N (since we required that when halting the work tape is left with a single value on it, and the head positions of both tapes return to the beginning, this configuration is unique).

This construction ensures that N accepts x if and only if there is a path from s to t in G, so L reduces to PATH.

To finish the proof we show that PATH reduces to CONN: Given a triple (G, s, t), we construct the graph H by taking G and adding edges from every vertex into s, and edges from t to every vertex.

It follows easily that if G has a path from s to t, then H is connected, and if G has no path from s to t, then neither does H, so it is not connected. □

So nondeterministic logarithmic space can be simulated in (deterministic) polynomial time. What about deterministic *space* simulations? When we tried to simulate nondeterministic time deterministically, we had to pay with an exponential blowup; in contrast, nondeterministic space is easy to simulate much more efficiently.

Theorem 6 (Savitch's theorem)**.** *Every* NL *problem can be decided by a deterministic TM that runs in space* $O((\log n)^2)$.

Proof. It is sufficient to show that PATH can be decided in space $O((\log n)^2)$. In fact we will design a TM that decides the more general question whether there is a path from s to t of length at most k in G. The machine will use space $O((\log k)(\log n))$, so when $n = k$ we obtain the desired consequence. Assume that k is a power of 2 and the vertex set of G is $\{1, \ldots, n\}$.

On input a graph G, vertices s, t, and a number k:
    If $s = t$ or there is an edge from s to t in G, accept.
    Allocate $\log n$ bits of work tape for a vertex index v.
    For $v := 1$ to n:
        Recursively test if there is a path from s to v of length $\leq k/2$.
        Recursively test if there is a path from v to t of length $\leq k/2$.
        If either path doesn't exist, try the next v.
        Otherwise, accept.
    Reject.

It is easy to see the algorithm is correct: There is a path from s to t of length at most k if and only if there are paths of length at most $k/2$ from s to v and from v to t for some v. What about the space complexity? Each level of recursion uses up $O(\log n)$ bits of work tape. Moreover, when running the recursion we need to store the positions of the work tape and input tape heads, which incur another $O(\log n)$ bits. Since the recursion runs for $O(\log k)$ levels, we obtain the desired space complexity. □

Therefore nondeterministic computations can be simulated deterministically (i) in polynomial time and (ii) in space $O((\log n)^2)$. However we do not know if there is a single simulation that runs both in polynomial time and $O((\log n)^2)$ space.

5 The Immerman-Szelepcsényi theorem

Finally, we show perhaps the most surprising way in which nondeterministic time and space appear very different. Recall our discussion of coNP: While we can easily verify that, say, a CNF formula has a satisfying assignment, we do not know in general how to verify there are no satisfying assignments. So it appears that NP $\neq$ coNP. In contrast, we will show that NL = coNL. So there is no essential difference between space-efficient proofs and refutations!

Theorem 7. NL = coNL.

To prove this theorem, it is sufficient to show that CONN $\in$ coNL, or equivalently, $\overline{\text{CONN}} \in$ NL. We design a logarithmic space NTM N that, on input a graph G, accepts if and only if G is *not* connected. It will be convenient to think of the (read-only) nondeterminism tape of N as containing a certificate, whose purpose it is to convince us that G is not connected.

Assume the vertices of G are $\{1, \ldots, n\}$. To certify that G is not connected, it is equivalent to check for some vertex v, the number c of vertices reachable from v is strictly smaller than n. The NTM will compute this number c and accept iff it is smaller than n.

What do we mean when we say that NTM N compute some number c? What if different numbers are computed on different computation paths? We won't give a formal definition, but what we mean is that all computation paths of N are either rejecting or contain the same number c on some specified portion of the work tape.

As an exercise, suppose that I wanted to convince you the number of vertices reachable from v is *at least* c. Our certificate for this task will consist of a list of c distinct vertices $z_1, \ldots, z_c$, ordered lexicographically, together with the list of vertices on some path (of length at most n) from v to z_i for each i. More precisely, the certificate is provided as $z_1, p_1, \ldots, z_c, p_c$ where p_i is the path from v to z_i. To verify the validity of the certificate, we need to check that there are at most c pairs in the list, $z_i < z_{i+1}$ for every i, and p_i is indeed a path from v to z_i in G. All these checks can be carried out in logarithmic space (with some "temporary variables" to remember the last values of i and z_i and the last vertex visited on the path p_i).

However, what we are really interested in knowing is that the number of vertices reachable from v is *at most* $n-1$, not at least something. This appears like a very different task! But it turns out that the two are related.

The trick is to compute the number c in stages. In stage k, the NTM will compute the number c_k of vertices reachable from v in *at most* k *steps*. Notice that $c = c_n$. So the certificate for c will contain the numbers $c_0, c_1, \ldots, c_n$, interspersed with some information that will convince us these values are indeed correct.

Initially, we have that $c_0 = 1$. Now assume that NTM N has already managed to compute c_{k-1}. We will show how it can then compute c_k.

To certify the value of c_k, we need two things: A *lower bound* that convinces us the number of vertices reachable from v in $\leq k$ steps is at least c_k, and an *upper bound* that convinces us this number is at most c_k. We already saw how we can certify a lower bound: The only difference here is that we need to check the length of all the paths p_i is at most k, and this can be done easily in logarithmic space. So we are left with the upper bound.

Given the value c_{k-1}, the certificate will contain a list of all vertices w in lexicographic order that are at distance at most $k-1$ from v, accompanied by a path that certifies the distance is at most $k-1$. By looking at the neighbors of each w, we can determine – and count – all the vertices u that are at distance at most k from v. However, this may not give us the correct value of c_k, because some of the vertices u may be counted multiple times! But when we look at u we can add a portion of the certificate that convinces us u has been visited before: If u was visited before, then there

must be some $w' < w$ such that w' connects to u and w' is at distance at most $k-1$ from v. So for each potential u, the certificate will contain a bit specifying if u is visited for the first time in round k, and if not, the identity of the vertex w' that certifies u was visited before (together with a certificate that w' is at distance at most $k-1$ from v).

Putting all these observations together, we obtain the following nondeterministic logarithmic space TM N for $\overline{\text{CONN}}$. This TM uses the nondeterministic subroutine path(G, v, w, k), which accepts iff there is a path from v to w in G of length at most k.

Algorithm N: On input a graph G with vertex set $\{1, \ldots, n\}$:
Guess a vertex v. (Supposedly, some vertex is unreachable from v).
Let $c := 1$. (c will contain the value c_{k-1}.)
For $k := 1$ to n:
Guess $c' \in \{1, \ldots, n\}$. (c' is supposed to equal c_k.)
Lower bound certificate for c':
For $i := 1$ to c':
Guess a vertex z lexicographically greater than the previous one, if one exists.
(Initially, any z is allowed.)
If path(G, v, z, k) rejects, reject.
Upper bound certificate for c':
Set $t := 0$. (t will be an upper bound on c'.)
For $i := 1$ to c:
Guess a vertex w lexicographically greater than the previous one, if one exists.
(Initially, any w is allowed.)
If path$(G, v, w, k-1)$ rejects, reject.
For every u such that (w, u) is an edge in G:
Guess if u was visited before (in the loop for i).
If yes, guess a vertex $w' < w$ such that (w', u) is an edge in G.
If path$(G, v, w', k-1)$ rejects, reject.
If not, set $t := t + 1$.
If $t > c'$, reject.
Set $c := c'$.
If $c < n$, accept.