$\mathbf{NC}^1$, Branching Programs, **PSPACE**, and Bottleneck Machines

Advanced Complexity, Homework Assignment

There are several parts in this homework assignment. They are *not* independent.

The solutions given here are only one among many. So don't think that your solution is wrong just because it is different. On the other hand, I'll take care of explaining the difficulties that we are faced with in each question, and the solutions are meant to give one way of getting around.

1 The Class $\mathbf{NC}^1$

The class $\mathbf{NC}^1$ is a curious class, defined in the same way as (uniform) $\mathbf{P/poly}$: it is the class of languages L that are decided by a family of circuits $\mathcal{C}_n$, $n \in \mathbb{N}$:

- whose depth is an $O(\log n)$—depth is the largest number of gates traversed from input to output;
- whose size is polynomial in n;
- that have *bounded fan-in*: each gate ($\wedge$, $\vee$, $\overline{\wedge}$, $\overline{\vee}$-gate) takes at most two inputs—the circuits we were considering in the lectures had unbounded fan-in, as each gate could have arbitrarily many inputs;
- and that are *uniform*, in the sense that there is a logspace Turing machine $\mathcal{M}$ that, on input x (of size n), computes $\mathcal{C}_n$.

1. In the above definition, why is the second requirement (that $\mathcal{C}_n$ be of polynomial size) redundant?

   We can garbage-collect the circuit, i.e., remove all wires that do not contribute to the result, in an intuitive sense. Define the wires that do contribute as the smallest set containing the output wire, and such that for every gate whose output contributes, all its inputs contribute.

Refine this: say that the output wire contributes with 0 delay, and that if i is a wire that contributes with delay d, and there is gate with output i, then all its wires contribute with delay $d+1$.

Then replace $\mathcal{C}_n$ by a circuit where all wires contribute, by eliminating the others and renumbering.

Since the depth of the (modified) $\mathcal{C}_n$ is at most $k \log n$ for some constant k, and for n high enough, one shows easily by induction on d that there are at most 2^d wires that contribute with delay d—the 2 comes from the bound 2 on the fan-in. So there are at most $2^0 + 2^1 + 2^2 + \ldots + 2^{k \log n} \leq 2^{k \log n + 1} = 2n^{k \log 2}$ wires in $\mathcal{C}_n$.

This is polynomial, and since (again, by bounded fan-in) the size of $\mathcal{C}_n$ is bounded by this times a constant times $\log n$ (for the size of the wire numbers), the size is polynomial.

A simpler reason is given by the fact that $\mathcal{C}_n$ is computed in logspace, therefore in polynomial time, and hence cannot be more than polynomial size.

2. Show that $\mathbf{NC}^1 \subseteq \mathbf{L}$. Hint: once you realize the naive algorithm you have written first takes more than logarithmic space, realize that given any logarithmic depth circuit $\mathcal{C}_n$, with output wire number q, then one can use logarithmic length bit strings to denote wires $i < q$ (instead of the number i itself): the empty string ϵ denotes q itself, while if w denotes a wire i, output of a gate G_i with two inputs i_1, and i_2, then $w0$ denotes i_1 and $w1$ denotes i_2. We shall call such bit strings w *paths*.

   Please use the notations: i_w to mean the wire denoted by path w; and op_w to denote the operator implemented by the gate G_{i_w}.

   Note that paths can be used to implement recursion stacks, as well.

   Given an input x of size n, one can compute $\mathcal{C}_n$ in logspace, by the uniformity assumption. The question is now to evaluate $\mathcal{C}_n[x]$ in logarithmic space, from input $\mathcal{C}_n$ and x—we shall conclude since the composition of two logspace computations is again logspace.

   Remember that a circuit is given by a netlist, where wire i only depends only on the values of wires j, $j < i$.

   We could evaluate the output wire q by calling $eval(q)$, where $eval(j)$ evaluates wire j as a function of (at most 2) calls to $eval(i)$ with $i < j$, recursively. This works because we can implement recursion with a stack, and because stack depth (=circuit depth) is logarithmic.

   But we must pay attention to only store constant-size data on the stack. In particular, the naive approach that implements $eval(i)$ by calling $eval(j)$ for all input gates j must store j on the stack, and is doomed.

   Instead, we use the idea given in the statement. This must be completed in some arbitrary way in the case where w denotes a wire i which is not the output of a

gate with two inputs. If it is a gate with one input, we let $w0$ *and* $w1$ *both denote this input. Otherwise, we let* $w0$ *and* $w1$ *denote wire* 0.

Given such a path w*, it is easy to compute the number* i_w *of the wire that* w *denotes, in logarithmic space: start with* $i := q$*, then read the successive bits of* w*, taking the left or right input wire to the corresponding gate (on the input tape). This needs two counters, one for* i *and one for its next value.*

The same algorithm also allows us to obtain the operator op_w *(*$\wedge$*,* $\vee$*,* $\overline{\wedge}$*,* $\overline{\vee}$*) that is computed by the gate whose output is* i_w*.*

We use a single counter for paths w*, and the structure of paths will allow us to use it as a stack as well, remembering directions (first/second input) we have taken in an ideal recursive algorithm. We also use a second stack of bits* res*, which remembers results of other branches of the ideal recursive algorithm. We store one bit in the finite control of the machine, but write this as a variable* val*.*

This goes as follows:

(a) Set $w := \epsilon$ *(start with an empty stack, we are trying to evaluate wire* $i_\epsilon = q$*)*

(b) Set $res := \epsilon$ *(stack of other results currently empty);*

(c) If op_w *is just true (i.e.,* $\wedge$ *with no input) then* $val := \top$

(d) If op_w *is just false (i.e.,* $\vee$ *with no input) then* $val := \bot$

(e) If op_w *has at least one input (and at most* 2*), then:*

i. Push 0 *onto* w*, i.e., set* $w := w0$*, and call oneself recursively: go to step 2c*

ii. We shall return here once we have evaluated wire i_{w0}*, and* w *will again contain what it contained at the beginning of step 2e.*
Push val *onto* res*: i.e., concatenate the bit in* val *at the end of the bit string* res*.*

iii. Push 1 *onto* w*, i.e., set* $w := w1$*, and make another recursive call: go to step 2c*

iv. We shall return here once we have evaluated wire i_{w1}*, and* w *will again contain what it contained at the beginning of step 2e. Moreover, the top of the* res *stack contains the value of wire* i_{w0}*.*
Pop the last bit of res*, compute the logical and, or, nand, or nor of this bit with* val *(depending on whether* op_w *is* $\wedge$*,* $\vee$*,* $\overline{\wedge}$*, or* $\overline{\vee}$*), and set* val *to the result thus obtained.*

v. Now we must return:

A. If $w = \epsilon$*, we are finished, and* val *contains the value of wire* $i_\epsilon = q$*: accept if* $val = \top$*, reject if* $val = \bot$*;*

B. If $w = w'0$*, then set* $w := w'$ *(pop the stack) and go to step 2(e)ii*

C. If $w = w'1$*, then set* $w := w'$ *and go to step 2(e)iv.*

Now this only takes logarithmic space, namely twice the depth of the circuit, plus the space needed to compute i_w*,* op_w *from* w*.*

2 Branching Programs

A *length n width k branching program* π (for short, an n, k-BP) is any non-empty finite sequence of *instructions* of the form `if` x_i `then` $R := f(R)$ `else` $R := g(R)$, where $0 \leq i < n$, and f and g are functions from $\{1, \ldots, k\}$ to $\{1, \ldots, k\}$. These are meant to work on a read-only bit string x of length n (or more) and with a unique read-write *register* R, taking its values in $\{1, \ldots, k\}$. The *size* of π is its number of instructions.

We assume the obvious semantics—x_i denotes bit i of input x, and the `if` test computes $f(R)$ if x_i is 1, $g(R)$ if x_i is 0. For example, the $9, 2$-BP:

$$\begin{array}{l}\texttt{if } x_7 \texttt{ then } R := (1\ 2)(R) \texttt{ else } R := R;\\ \texttt{if } x_4 \texttt{ then } R := R \texttt{ else } R := (1\ 2)(R);\\ \texttt{if } x_8 \texttt{ then } R := (1\ 2)(R) \texttt{ else } R := R;\end{array}$$

will test whether exactly one of x_7, $\neg x_4$, and x_8 is true. If so the final value of R will be 1 if started with $R = 2$, and 2 if started with $R = 1$. Otherwise, the final value of R will be the same as when we started the program. (The map (1 2) is the permutation that swaps 1 and 2).

In general, a *permutation* is any bijective map from $\{1, \ldots, k\}$ to $\{1, \ldots, k\}$. A *cycle* on $\{1, \ldots, k\}$ is any permutation of the form $(m_1\ m_2\ \ldots\ m_k)$ (with m_1, m_2, $\ldots$, m_k pairwise distinct and between 1 and k) that sends m_1 to m_2, m_2 to m_3, $\ldots$, m_{k-1} to m_k, and m_k to m_1. E.g., (1 2 3 4 5) is a cycle (this is *not* the identity) when $k = 5$, as well as (1 3 5 4 2) or (5 4 3 2 1).

It is important to note that every cycle $\sigma = (m_1\ m_2\ \ldots\ m_k)$ yields a unique *associate* permutation σ', which maps each $i \in \{1, \ldots, k\}$ to m_i. E.g., the associate of the cycle (1 2 3 4 5) is the identity map (which is not itself a cycle).

An n, k-BP is a *permutation branching program* (an n, k-PBP) iff the functions (f, g, from $\{1, \ldots, k\}$ to $\{1, \ldots, k\}$) used in its instructions are all permutations.

Given an input x of length n, an n, k-BP (resp., n, k-PBP) π defines a map (resp., a permutation) from $\{1, \ldots, k\}$ to $\{1, \ldots, k\}$, which sends the initial value of R to the value it has at the end of the program. Call f_π this map (resp., permutation).

We say that a language L of bit strings of length n (i.e., the length is fixed) is *cycle-recognized* by an n, k-PBP π iff there is a cycle σ over $\{1, \ldots, k\}$ such that:

- For every $x \in L$, $f_\pi = \sigma$;
- For every $x \notin L$ of length n, f_π is the identity map.

If this is so, we say that L is cycle-recognized by π *with output* σ. Notice that this is well-defined, since σ, as a cycle, cannot be the identity. (A cycle has no fixpoint.)

Our first move is to show that which cycle σ we choose in defining cycle-recognition is irrelevant.

3. Let π be an n, k-PBP cycle-recognizing L with output σ, and τ be any cycle over $\{1, \ldots, k\}$. Build another n, k-PBP π', of the same size as π, that cycle-recognizes

L with output τ. Hint: first show that there is a permutation θ such that $\tau = \theta\sigma\theta^{-1}$ (where we write e.g. $\theta\sigma$ for $\theta \circ \sigma$, for short); this can be built using associate permutations.

Existence of θ. Let $\sigma = (m_1\ m_2\ \ldots\ m_k)$, $\tau = (n_1\ n_2\ \ldots\ n_k)$. The associate maps $\sigma' : i \mapsto m_i$ and $\tau' : i \mapsto n_i$ must be bijections, since σ and τ are cycles. Let $\theta = \tau'\sigma'^{-1}$.

Writing succ for the function that maps i to $i+1$ if $i \neq k$, and to 1 otherwise (successor modulo k, essentially), we have $\sigma\sigma' = \sigma' succ$ and $\tau\tau' = \tau' succ$. So $\theta\sigma\theta^{-1} = \tau'\sigma'^{-1}\sigma\sigma'\tau'^{-1} = \tau'\sigma'^{-1}\sigma' succ\tau'^{-1} = \tau' succ\tau'^{-1} = \tau\tau'\tau'^{-1} = \tau$.

Now, obtain π' from π by replacing each permutation f (or g) in each instruction by $\theta f\theta^{-1}$. Since $f_\pi(x)$ is a composition $f_1 f_2 \ldots f_p$ of permutations from π, $f_{\pi'}(x)$ will be the corresponding composition $\theta f_1\theta^{-1}\theta f_2\theta^{-1}\ldots\theta f_p\theta^{-1} = \theta f_\pi(x)\theta^{-1}$.

In particular, if $x \in L$, then $f_{\pi'}(x) = \theta\sigma\theta^{-1} = \tau$, while if $x \notin L$, then $f_{\pi'}(x) = \theta\mathrm{id}\theta^{-1} = \mathrm{id}$, whence the claim.

4. Let L be a language of bit strings of length n, and $\overline{L}$ be its complement inside the set of length n bit strings. Given an n, k-PBP π that cycle-recognizes L, build another one, $\overline{\pi}$, of the same size, but that cycle-recognizes $\overline{L}$.

   Assume π cycle-recognizes L with output σ. Then replace the last instruction `if` x_i `then` $R := f(R)$ `else` $R := g(R)$ *by* `if` x_i `then` $R := (\sigma^{-1}f)(R)$ `else` $R := (\sigma^{-1}g)(R)$. *This cycle-recognizes $\overline{L}$ with output σ^{-1}.*

5. Let $k = 5$, $\sigma_1 = (1\ 2\ 3\ 4\ 5)$, $\sigma_2 = (1\ 3\ 5\ 4\ 2)$. It is easily checked that the commutator $\sigma_1\sigma_2\sigma_1^{-1}\sigma_2^{-1}$ of these two cycles is the cycle $(1\ 4\ 3\ 5\ 2)$.

   Deduce *Barrington's Lemma*: if L_1 is cycle-recognized by an $n, 5$-PBP π_1 of size t_1 and L_2 is cycle-recognized by an $n, 5$-PBP π_2 of size t_2, then $L_1 \cap L_2$ is cycle-recognized by some $n, 5$-PBP of size $2(t_1 + t_2)$.

   We can assume that π_1 recognizes L_1 with output σ_1, and that π_2 recognizes L_2 with output σ_2, by Question 3. By Question 3 again, L_1 is also cycle-recognized by an $n, 5$-PBP π_1^{-1} of size t_1 with output σ_1^{-1}, and L_2 is also cycle-recognized by an $n, 5$-PBP π_2^{-1} of size t_2 with output σ_2^{-1}.

   Then the concatenation $\pi_1\pi_2\pi_1^{-1}\pi_2^{-1}$ cycle-recognizes $L_1 \cap L_2$ with output $\sigma_1\sigma_2\sigma_1^{-1}\sigma_2^{-1}$. Indeed, if $x \in L_1 \cap L_2$, then $f_{\pi_1\pi_2\pi_1^{-1}\pi_2^{-1}}(x)$ is the composite $\sigma_1\sigma_2\sigma_1^{-1}\sigma_2^{-1}$. If $x \in L_1$ but $x \notin L_2$, then $f_{\pi_1\pi_2\pi_1^{-1}\pi_2^{-1}}(x)$ is $\sigma_1\mathrm{id}\sigma_1^{-1}\mathrm{id} = \mathrm{id}$, and similarly if $x \in L_2$ but $x \notin L_1$. Finally, if $x \notin L_1$ and $x \notin L_2$, then $f_{\pi_1\pi_2\pi_1^{-1}\pi_2^{-1}}(x) = \mathrm{idididid} = \mathrm{id}$.

Until now, we were recognizing languages of fixed length n. Say that a language L (of words of arbitrary length n) is *decided* by a family $(\pi_n)_{n\in\mathbb{N}}$ of n, k-PBPs π_n (of fixed width

k, but varying n) if and only if, for every input x, writing n for the size of x, if $x \in L$ then $f_{\pi_n}(x)(1) = 2$ and if $x \notin L$ then $f_{\pi_n}(x)(1) = 1$.

Such a family is *uniform* iff there is a logspace Turing machine that, given any input of size n (i.e., n written in unary), computes π_n.

6. Using the previous questions, show that every language in $\mathbf{NC}^1$ is decided by a uniform family of PBPs of polynomial size, and of width 5.

   Assume $L \in \mathbf{NC}^1$. So L is decided by a uniform family of bounded fan-in, polynomial size, logarithmic depth circuits $\mathcal{C}_n$. The idea is to convert $\mathcal{C}_n$ (say of depth $d = O(\log n)$), recursively, into a branching program of size at most 4^d, using Question 5 to implement logical ands (this at most quadruples the size, compared to circuits of depth at most $d-1$) and Question 4 to implement logical negations. Disjunction $A \vee B$, being equivalent to $\neg(\neg A \wedge \neg B)$, is reduced to the previous cases.

   However, the difficulty is in showing uniformity*, i.e., that the PBP can be built using logspace only.*

   Using the fact that logspace reductions compose, we first produce $\mathcal{C}_n$ in logspace, then convert $\mathcal{C}_n$ to another circuit that only uses nand ($\overline{\wedge}$) gates for example. (Rewriting ands as negations of nands, ors as nands of negations, and negations as unary nands.) This can be done in logspace, and increases the depth by a factor at most 2.

   So we assume that only gates in $\mathcal{C}_n$ are nand gates, of fan-in 0, 1, or 2.

   Given $\mathcal{C}_n$, we can convert it into an n, k-PBP that cycle-recognizes the subset L_n of those words in L of length n, say with output σ_1. This is by induction on the depth d of $\mathcal{C}_n$. As in Question 2, we recurse on paths from the root through $\mathcal{C}_n$, maintaining a logspace stack.

   However, there is a difficulty: we shall sometimes need to build PBPs with output different from σ_1 (i.e., to use Question 3, and we cannot easily do this by outputting a PBP with output σ_1 and then applying the transformation of Question 3: although the latter is logspace, and logspace reductions compose, this composition needs to be done inside recursive calls. We have a similar problem with negations, where we again need to modify a PBP we just constructed: although we only need to swap the last then and else branches, we cannot do so because we cannot read from the output tape).

   So we instead build a recursive function $cvt(w, \sigma, neg)$ that builds an n, k-PBP that cycle-recognizes the sub-circuit of $\mathcal{C}_n$ rooted at w (if $neg = 0$, or its negation if $neg = 1$), with prescribed output σ. The recursion stack cannot be implemented inside w alone, since we need to remember the various σ's and neg's submitted to each recursive call. But each of these has constant size (5 numbers, of size $\log 5$, plus one bit).

If op_w is nand with no input (false), then $cvt(w, \sigma, neg)$ must always return σ if $neg = 1$ (true) or id *if $neg = 0$ (false): define it as* `if` x_0 `then` $R := \sigma(R)$ `else` $R := \sigma(R)$ *if $neg = 1$, and as* `if` x_0 `then` $R := \mathrm{id}(R)$ `else` $R := \mathrm{id}(R)$ *if $neg = 0$. (Technically, this causes a problem if $n = 0$, and the statement of the problem is, accordingly, wrong in a formal sense. This is not serious, though.)*

If op_w is nand with 1 input (negation), then $cvt(w, \sigma, neg)$ is simply $cvt(w0, \sigma, \neg neg)$.

Finally, if op_w is nand with 2 arguments, then we use Question 5. First we use Question 3, and find (this can be precompiled) θ such that $\sigma = \theta\sigma_1\sigma_2\sigma_1^{-1}\sigma_2^{-1}\theta^{-1}$. Then $cvt(w, \sigma, neg)$ is the concatenation of $cvt(w0, \theta\sigma_1\theta^{-1}, 0)$, $cvt(w1, \theta\sigma_2\theta^{-1}, 0)$, $cvt(w0, \theta\sigma_1^{-1}\theta^{-1}, 0)$, and $cvt(w1, \theta\sigma_2^{-1}\theta^{-1}, neg)$. (Note that the first three recursive calls must have the neg bit set to 0, since negation—if any—only modifies the last instruction of the PBP, which is necessarily the last instruction of the fourth PBP.)

Now $cvt(\epsilon, \sigma, 0)$ will produce, in logspace, an $n, 5$-PBP that cycle-recognizes the language of words of length n in L, with output σ. Taking σ so that $\sigma(1) = 2$ (e.g., $\sigma = \sigma_1$) fits the bill.

And finally, the size of the PBPs is at most $4^{k \log n}$ where $k \log n$ is the depth of $\mathcal{C}_n$, and this is polynomial since $4^{k \log n} = n^{k \log 4}$.

7. Conclude that $\mathbf{NC}^1$ is exactly the class of languages decided by uniform families of PBPs of polynomial size and width 5. Hint: split PBPs in two, recursively.

It remains to show that any language L that is decided by a uniform family $(\pi_n)_{n \in \mathbb{N}}$ of $n, 5$-PBPs is in $\mathbf{NC}^1$. Build π_n in logspace. Say π_n has length ℓ. Up to adding extra instructions of the form `if` x_0 `then` $R := \mathrm{id}(R)$ `else` $R := \mathrm{id}(R)$*, we can assume ℓ is of the form 2^m, for some $m = O(\log p(n)) = O(\log n)$. (This also only needs an additional logspace counter.)*

Assume π_n cycle-recognizes with output, say σ_1.

Given an n, k-PBP π that arises as a (contiguous) subsequence in π_n, of length 2^m, we convert it into a circuit $\mathcal{C}[\pi]$ that takes the n input bits of x, the input bits of R, and has output bits represent the final value of R when π terminates. For ease of presentation, instead of encoding R in binary, using $\lceil \log k \rceil$ bits, we encode it in unary using k bits.

- *(Base Step) If $m = 0$ (size 1), then π is of the form* `if` x_i `then` $R := \tau_1(R)$ `else` $R := \tau_2(R)$ *for some permutations τ_1 and τ_2. Build a k-input wire, k-output wire circuit for τ_1, another one for τ_2 (these are just collections of k circuits, one for each output bit; and they are of constant size). Plug the jth input of these two circuits together, for each $j = 1, \ldots, k$: these are the R-inputs. Given that the jth output wires of the two circuits are, say, $j/1$ and $j/2$, the jth output wire of the circuit for π should be "if x_i is true then $j/1$ else $j/2$", which can be implemented by a* selector *circuit, say $(x_i \wedge j/1) \vee (\neg x_i \wedge j/2)$.*

- *(Recursion Step) If $m \geq 1$, then π is the concatenation of two PBPs π_1 and π_2 of size 2^{m-1}. Build $\mathcal{C}[\pi_1]$ and $\mathcal{C}[\pi_2]$. It is tempting to build $\mathcal{C}[\pi]$ by plugging the k output wires of $\mathcal{C}[\pi_1]$ with the k input wires of $\mathcal{C}[\pi_2]$, but this would eventually produce a circuit of polynomial, not logarithmic, depth.*
  Instead, realize that the composition of the two circuits can be obtained as a disjunction, over all 5 possible values R of $\mathcal{C}[\pi_1]$, of the circuit $\mathcal{C}[\pi_2]$ with its inputs connected to the binary representation of R.
  I.e., build 5 copies $\mathcal{C}_1, \ldots, \mathcal{C}_5$ of $\mathcal{C}[\pi_2]$ (by calling ourselves recursively for each: we do not have enough space to store a circuit to copy from), connect the input wires of each $\mathcal{C}_i$ to constant gates representing the value i in binary, $i \in \{1, \ldots, 5\}$; implement "if $R = 1$ then w_1 elsif $R = 2$ then w_2 elseif ... elsif $R = 5$ then w_5" (a function of R, $w_1, \ldots, w_5$) as a constant-size circuit with inputs k bits for R and the $5k$ wires $w_1, \ldots, w_5$, and with k output wires: then connect the w_i wires to the output of $\mathcal{C}_i$, and the R wires to the output of $\mathcal{C}[\pi_1]$.
  The depth of the resulting circuit is then the max of the depths of $\mathcal{C}[\pi_1]$ and $\mathcal{C}[\pi_2]$, plus a constant.

So the depth of the final circuit is logarithmic in the size of the PBP π, hence of size $O(\log n)$.

Implementing this recursively, as usual, is not logspace: we need a stack of $m = O(\log p(n))$ entries, remembering the starting position and the end position of π inside π_n, totaling $O(\log^2 n)$ space.

However, we do not need to keep these entries at all. Imagine for now that we did not need to make 5 copies of $\mathcal{C}[\pi_2]$ in the Recursion Step. (This will make the idea clearer. We shall revert to the 5 copies later.) The positions that we kept above in π_n only serve to read a given instruction of the PBP π_n in the Base Steps. But then, the structure of the recursion has the effect that the first Base Step we run into will read the first instruction, the second Base Step will read the second one, then the third instructions in π_n, etc., in sequence. So the Base Step should just additionally move its head to the next instruction in the PBP π_n.

Since we need to call ourselves recursively 5 times to compute 5 copies of $\mathcal{C}[\pi_2]$ in the Recursion Step, the situation is a bit more complex. We use a similar trick as in Question 2, and build a function $cvt(w)$, where w is a string of length at most $m = O(\log p(n))$, which will act both as an indication of what subsequence of π_n should be compiled into a circuit, and as recursion stack, but where w is now a string over $\{0, 1, \ldots, 5\}$: 0 will be to recurse and compute $\mathcal{C}[\pi_1]$ in the Recursion Step, the other values will be to compute the 5 copies of $\mathcal{C}[\pi_2]$. Let $bin(w)$ be w where all non-zero numbers have been replaced by 1.

Now the Base Step will be as above, except it should move to the next instruction in π_n if and only if the last letter of w (which should be of length exactly m at this point) is either 0 or 5.

Formally, we also need to avoid storing wire numbers in local variables, since

they would each take $O(\log n)$ space, and we would need $O(\log n)$ of them on the recursion stack, for a total of $O(\log^2 n)$ space. A solution is to preassign *the wire numbers by some easy algorithm: we can then recompute the desired wire numbers instead of storing them.*

Let us for example agree that the input wires for the Base Step circuits (at position $bin(w)$, where $|w| = m$) are numbered $w01, \ldots, w05$ (read as numbers in base 6), and its ouput wires are numbered $w11, \ldots, w15$, or some similar scheme.

We now define $cvt(w)$ as $cvt_1(w, 0)$ where $cvt_1(w, R)$ will compute the desired circuit if $R = 0$, and compute the desired circuit with inputs fixed to the value R otherwise (we need to do so, so as to be able to list wires with low numbers before wires with higher numbers, as required in our formal definition of circuits). We first initialize a global variable i to 0 (this is the instruction counter in π_n, and will run from 0 to $p(n) = 2^m$, hence will only take logarithmic space).

- *(Base Step) If $|w| = m$ (counting lengths with binary counters uses only logspace), then consider instruction number i in π_n, say* `if` $x_{i'}$ `then` $R := \tau_1(R)$ `else` $R := \tau_2(R)$*, for some permutations τ_1 and τ_2. Build a k-input wire, k-output wire circuit for τ_1, another for τ_2, of constant size, with preassigned numbers. Plug the jth input of these two circuits together, for each $j = 1, \ldots, k$: these are the R-inputs; if $R \in \{1, \ldots, 5\}$, additionally connect these inputs to the binary representation of R.*
  Given that the jth output wires of the two circuits are, say, $j/1$ and $j/2$, the jth output wire of the circuit for π is "if $x_{i'}$ is true then $j/1$ else $j/2$".
  Now, if the last letter of w is 0 or 5, then increment i.
- *(Recursion Step) If $|w| < m$, then π is the concatenation of two PBPs π_1 and π_2. Call $cvt(w0, R)$, $cvt(w1, 1)$, $\ldots$, $cvt(w5, 5)$ recursively (using w as a stack, as we did in Question 2). Since circuits have preassigned output wire numbers, we do not require cvt to return any wire number, and accordingly we do not store them.*
  Now build the constant-size circuit "if $R' = 1$ then w_1 elsif $R' = 2$ then w_2 elseif $\ldots$ elsif $R' = 5$ then w_5", and connect its k-bit inputs R' to the outputs of $cvt(w0, R)$ (retrieving their preassigned numbers from w), and each w_j to the (preassigned) outputs of $cvt(wj, j)$.

So one can compute $\mathcal{C}[\pi_n]$ in space logarithmic in n. We cap all this by a test whether it implements (for fixed x) the permutation σ_1 or id*. It is enough to test whether the implemented permutation maps 1 to 2: so plug the k input bits for R (those that are not connected to x) to the binary representation of 1—this means we actually compute $cvt(\epsilon, 1)$, not $cvt(\epsilon, 0)$—and test whether the k output wires collectively represent the number 2, i.e., whether the second bit is set.*

3 PSPACE and Bottleneck Machines

8. Show that, given a language L decided by an alternating Turing machine $\mathcal{M}$ that works in polynomial time $p(n)$, one can build a circuit $\mathcal{C}_n$ of polynomial depth, with n input bits, such that $\mathcal{C}_n[x]$ evaluates to 1 iff x is in L (for any bit string x of length n).

   Let $\mathcal{M}$ be an alternating Turing machine that works in polynomial time $p(n)$ (hence in space at most $p(n)$ plus some constant as well). Fix the size n of the input x.

   A run of this machine is not a sequence of configurations but a tree of configurations, and it is accepting iff all its leaves are in the accepting state. We may assume without loss of generality that:

   - *the length of each branch is exactly $p(n)$: as in Cook's Theorem, we may modify the machine so that it loops, and does not stop, on an accepting state; now x will be in L iff all branches of the run have reached the accepting state at time $p(n)$;*
   - *the tree is binary, in the sense that every non-leaf vertex has exactly two successors; indeed, one can reduce universal non-determinism to two-way universal non-determinism, ensuring that every non-leaf vertex has at most two successors; and one can convert any deterministic configuration (only one successor) into a universal non-deterministic one that has the same two successor configurations.*

   A position in this tree is now given by a word $w \in \{0,1\}^*$ of length at most $p(n)$.

   Encode each configuration as in Cook's Theorem, and assume the letters are just bits (up to encoding, taking a constant factor). Create wires for each position w in the tree and each position $j \leq p(n)$ in the configuration at position w.

   The value at coordinates (wc, j) ($c \in \{0,1\}$) depends on a constant number of bits at coordinates $(w, j-a)$, $(w, j-a+1)$, $\ldots$, $(w, j+b)$ for some constants $a, b > 0$. We can compile this dependency as a (constant size) circuit connecting wires $(w, j-a)$, $(w, j-a+1)$, $\ldots$, $(w, j+b)$ to (wc, j).

   Do this for all w, c, j. This requires a constant number of registers taking logarithmic space (j, plus counters holding intermediate values) and one taking polynomial space (w).

   Leave the wires (ϵ, j) where j is a position in the input (roughly the first n bits, up to a constant number of bits at the beginning representing the initial state q_0) as inputs to the whole circuit, and connect the other wires (ϵ, j) to true or false depending on the bit representation of q_0 (at the left end) and of spaces (to the right of x).

   Assuming that $\mathcal{M}$ terminates after first putting back the head in position 0, one can check that branch w (of length $p(n)$) ended up accepting by building a constant-size circuit taking as input the first bits of the last configuration $(p(n), 0)$, $(p(n), 1)$,

..., $(p(n), c)$ (for some constant c) and outputs 1 if these bits denote the letter that represents the acceptance state. Let (w) be the output wire of this circuit.

We now check that all branches end up accepting by taking the logical and of the $2^{p(n)}$ wires (w), w of length $p(n)$. This can be done by a depth $p(n)$ stack of binary and gates.

9. Using the results above, show that every language L in **PSPACE** is decided by a so-called *bottleneck Turing machine* $\mathcal{M}$. Such a machine runs exponentially many phases in succession, phase 0 through $2^{p(n)} - 1$, where p is a fixed polynomial, but requires incredibly low space—and its memory gets almost completely erased regularly, if this were not enough.

   In each phase, the machine starts with access to:

   - the input tape (with the same input x, of size n, for all phases),
   - a $p(n)$ bit read-only counter tape holding the current phase number,
   - a read-write register R, holding a value in $\{1, 2, 3, 4, 5\}$,
   - a fixed number of work tapes, which are empty at the beginning of the phase.

   Whenever phase i starts, the machine is allowed to do some logspace computation using the above input (space refers to the work tapes), and write back a new value for R. Then the work tapes are entirely erased, the phase number counter is incremented, and phase $i + 1$ starts (unless $i = 2^{p(n)} - 1$). We run all phases 0, 1, ..., $2^{p(n)} - 1$, with R starting as 1 (in phase 0). Once the last phase finishes, the machine accepts if $R = 2$, rejects otherwise. (Note the similarities with PBPs.)

   Since L is in **PSPACE** $=$ **AL**, *there is an* **AL** *machine $\mathcal{M}$ deciding L. By Question 8, we can build a circuit $\mathcal{C}_n$ that will decide whether $x \in L$ for all x of size n.*

   Up to a size blowup, $\mathcal{C}_n$ is an $\mathbf{NC}^1$ *circuit. The question is now to convert a circuit, but of polynomial, not logarithmic, depth, into a big PBP.*

   We cannot produce the PBP in logarithmic space—we can only produce output of polynomial size, and the PBP has exponential size. Moreover, the circuit $\mathcal{C}_n$ itself has exponential size. But given the phase number i, we can produce the ith instruction in the desired PBP, and this is all that we need to do.

   So assume $\mathcal{C}_n$ is built using only binary nand gates (unary ones, i.e., negations $\neg x$ are encoded as the nand of x and x), and one true node. We run the procedure of Question 6, except that we only keep the ith instruction. This also implies that we only need to make one of the 4 recursive calls to cvt.

   I.e., assume $\mathcal{C}_n$ has depth $p(n)$. Then the procedure of Question 6 produces a program of size at most $4^{p(n)}$. To simplify things, we have it produce a program of size exactly *$4^{p(n)}$. Since we have no 1-ary nand any longer, it suffices to modify the*

definition of $cvt(w, \sigma, neg)$ when op_w is a nand with no argument, and $|w| < p(n)$ by still having it recurse 4 times, say computing a logical and of itself four times (assuming $w0$ and $w1$ refer to the same node as w, which is easy).

Now cvt must be guided by the integer i, which we now read (as an integer in binary, between 0 and $4^{p(n)} - 1$) as a $2p(n)$-bit string: $cvt'_i(w, \sigma, neg)$ returns the ith instruction in the PBP of Question 6, and w is a prefix of i (read as a bit string). Concretely, if $|w| < p(n)$ then i is of the form wb_1b_2w', where b_1, b_2 are two bits, and then $cvt'_i(w, \sigma, neg)$ equals (assuming we are in the binary nand case; the other cases are easy but tedious modifications, considering the above paragraph):

- *$cvt'_i(w0, \theta\sigma_1\theta^{-1}, 0)$ if $b_1 = b_2 = 0$;*
- *$cvt'_i(w1, \theta\sigma_2\theta^{-1}, 0)$ if $b_1 = 0$, $b_2 = 1$;*
- *$cvt'_i(w0, \theta\sigma_1^{-1}\theta^{-1}, 0)$ if $b_1 = 1$, $b_2 = 0$;*
- *$cvt'_i(w1, \theta\sigma_2^{-1}\theta^{-1}, neg)$ if $b_1 = b_2 = 1$.*

Now this is tail-recursive (whenever cvt'_i calls itself recursively, it is the last thing it will do), so we do not need a recursion stack: we replace recursive calls by gotos. So no stack space is involved at all.

Moreover, as w is always a prefix of i, it is enough to keep the length of w, from which we can easily compute the next bits b_1 and b_2 to read from i.

The other difficulty is that the procedure of Question 6 starts from a circuit $\mathcal{C}_n$, but that now $\mathcal{C}_n$ has exponential size. We have taken care to build it using polynomial space only in Question 8: the largest piece of data there is the path w in the run of the alternating Turing machine. This is still not enough, since composing the computation of cvt'_i above with this polynomial space computation would be polynomial space, not logspace.

However, the only thing that takes more than logarithmic space is the word w denoting the position in the run. But this is the same word as the word w considered above, which was encoded as the phase number.

10. Conversely, show that any language L decided by a bottleneck Turing machine is in **PSPACE**.

A bottleneck Turing machine is nothing but a logspace machine $\mathcal{M}$ that given input (x, i, R_i) (actual input x, phase number i, value of register R) will compute the new value $R_{i+1} = \mathcal{M}(x, i, R_i)$ of register R at the end of the phase.

It suffices to simulate the bottleneck machine $\mathcal{M}$ deciding L in the obvious way: reserve a tape holding the phase counter, store R in the control state or on some specific tape, and run through the phases, incrementing the phase counter and erasing the other working tapes at each phase change.

This all works in polynomial space, the largest space being taken by the phase counter.

It follows that **PSPACE** is exactly the class of languages that are decided by bottleneck Turing machines. This is the *Cai-Furst Theorem* (1991).