Complexity & Advanced Algorithms

Siddharth Bhat

Contents

Chapter 1

NLogSpace-completeness

1.1 Co-NLogSpace

$L \in$ Co-NLogSpace $\equiv L^c \in$ NLogSpace. That is, complement the language L. if L^c is in NLogSpace, then $L \in$ Co-NLogSpace.

We intuitively believe that NP $\neq$ Co-NP. However, we can show that NLogSpace = Co-NLogSpace.

$$\text{PATH} = \{\langle G, u, v\rangle \mid \textbf{exists} \text{ path between vertices } (u, v)\}$$
$$\overline{\text{PATH}} = \{\langle G, u, v\rangle \mid \textbf{no} \text{ path between vertices } (u, v)\}$$

We assume that $\overline{\text{PATH}}$is co-NL-Complete.

If we show that $\overline{\text{PATH}}$is in NLogSpace, then every problem in co-NLwill be in NL

1.1.1 Solving $\overline{\text{PATH}}$in NL

$$V_R \equiv \{\text{set of vertices reachable from } u\}$$
$$V_{NR} \equiv \{\text{set of vertices } \textbf{not} \text{ reachable from } u\}$$

Sid confusion, why can't we use PATH as a subroutine: When we have an NDTM, we cannot *observe that the NDTM returns a* 0. We can *observe if an NDTM succeeds*, but there are weird paths and exponential number of paths where the NDTM does not return a 0? But if this is true, then how is PATH NL-complete? I am very confused.

To represent V_R and V_{NR}, we use 1 bit per vertex (since V_R and V_{NR} are disjoint), so total space is V.

Assume we know $|V_R|$. In this case, we can check whether v is unreachable from u — Enumerate all vertices. If they are reachable from u, bump up a counter. If we don't hit v till the counter gets to $|V_R|$, then what we know that is v is unreachable.

However, if v were reachable from u, then as we enumerate, we would find v as we were going through all vertices (we would not hit V_R unless we visit v).

This is important, because in an NDTM, if *any* of the paths accept, then we accept.

$$V_R = \cup_i V_R(i)$$
$$V_R(0) = \{u\}$$

to compute $cur \in_? V_R(i+1)$, first **recompute** that $pred \in V_R(i)$, and then check that $(cur, pred) \in E(G)$. We cannot **store** $V_R(i)$, since we don't have enough space.

eventually we will reach $V_R(|V|)$, where we stop.

We can compute $|V_R| = \sum_i |V_R(i)|$. We compute $|V_R(i)|$ by checking over each vertex it's membership into $V_R(i)$. And if it does, we bump up our counter.

Reference: Read `Sipser-Chapter 8`

```
def belongs(G, i, startv, endv, curv):
    """Check if curv belongs to V_R(i)"""
    if i == 1:
        return startv == curv
    else:
        # log(V)
        for pred in G.vertices:
            # This can use a modified version of PATH that stores lengths?
            if small_belongs(G, i - 1, startv, endv, pred):
                if isneighbour(pred, curv):
                    return True

        return False

def countcard(G, startv, endv):
    """ Count the cardinality of V_R"""
    card = 0
    # log(V)
    for i in len(G.vertices):
        # this is also log(V)
        for curv in G.vertices:
            if small_belongs(G, i, startv, endv, curv):
                card += 1
    return 1
```

1.2 Oracles

For all inputs w of length $|w| = n$, there exists a **single** advice (a_n is allowed to be a single string that is polynomial in n). So, $a : \mathbb{N} \to \Sigma^*$, and the advice of a given input w is $a(|w|)$.

1.2.1 $\mathtt{P}^{\mathtt{poly}}$

$L \in \mathtt{P}^{\mathtt{poly}}$ if there is a polynomial time turing machine M which takes two inputs — a string $x \in \Sigma^*$, and an advice $a_n \in \Sigma^*$, such that for all inputs w such that $|w| = n$, then there exists a polynomial $p(n)$ with $|a_n| \leq p(|w|)$.

We force it to be polynomial in the word-length, because things like a lookup table take exponential space in the word-length (number of strings of length n is 2^n).

We can see that the advice is somewhat "hardwired" into the machine given the input length (since $a : \mathbb{N} \to \Sigma^*$). So, we have a sequence of machines $M_i : \mathbb{N} \to \{\text{Turing machines}\}$, and we instantiate the machine $M_{|w|}$ to check if $|w| \in L$.

NP is allowed to have a *varying witness*, while $\mathtt{P}^{\mathtt{poly}}$will have the *same* advice.

We don't even need to know if the advice string should be able to be found in polynomial time.

1.2.2 $\mathtt{P}^{\mathtt{poly}}$contains non-recursive languages

Chapter 2

Advice & Time Hierarchies

2.1 $\mathtt{P}^{\mathtt{poly}}$

This class could possibly be bigger than P.

In NP , witnesses are different for each string. In $\mathtt{P}^{\mathtt{poly}}$, witnesses are fixed for strings of a given length.

The advice string need to even be found in polynomial time!

Recursive language: Halts on all inputs with yes/no Recursively enumerable: Halts and returns yes on inputs which belong to the language. On inputs that do not halt, undefined behavior.

2.2 Unary language that is non-recursive

L is a unary language $\equiv L \subseteq 1^*$

Theorem 1 *Every unary language is decidable by* P^{poly}

Proof. let L be a unary language.

Since the only characteristic of a string in a unary language is its length, for any given length, there is *at most one string of that length* in L. So, we can index the set L by the string lengths! Hence, the advice function allows us to build up a lookup table for *any* unary language.

We construct the advice function $a_L : \mathbb{N} \to \{0,1\}$ be such that $a_L(n) =$ does 1^n belong to L?. Now, let M decide L as follows: $M(str) = a(|str|)$. Since we don't need to build a (it's an oracle we take for granted, the proof is done).

Theorem 2 P^{poly}*contains non-recursive languages.*

Proof. Let $L_{nr} \subset \{0,1\}^*$ be a nonrecursive language. We define $L_w = \{1^{\#w} \mid w \in L_{nr}\}$, which is a unary language. A string $1^k \in L_w$ acts as a witness for the existence of some string $w \in L_{nr}$ as the lex-ordering-position of the string w.

Example of # evaluated on some strings
$\#0 \to 0$
$\#1 \to 1$
$\#00 \to 3$
$\#01 \to 4$
$\#100 \to 5$
...

L_{nr} has now been reduced to L_u, since the mapping with # is a *bijection* . Also, L_u can be decided by $\mathtt{P}^{\mathtt{poly}}$. Hence, L_u can decide nonrecursive languages.

Question: Is the set $\{0,1\}^*$ countable? It doesn't feel like it is!

2.2.1 Sparse language

A **sparse language** is one where the number of strings of length n is bounded by a polynomial. $|L \cap \{0,1\}^n| \leq p(n)$.

Idle thought: Is there a classification theorem for sparse languages? "sparse-complete"

We study the relationship between NP and $\mathtt{P}^{\mathtt{poly}}$, using sparse languages.

2.2.2 Cook reduction

A language L_1 cook reduces to a language L_2 if there is a polynomial-time turing machine M_{L_1} that recognizes L_1 given oracle access to L_2.

The machine M_{L_1} Can query membership to L_2 multiple times (polynomial) before deciding if a string $w \in_? L_1$.

Lemma 1 *If L_1 Cook-reduces to L_2 and $L_2 \in P$, then $L_1 \in P$.*

Proof. L_1 is decided by a polynomial-time turing machine M_{L_1}, so it can make at most polynomial queries to L_2. Since $L_2 \in P$, There exists a polynomial-time turing machine M_{L_2} which solves the membership query.

The total running time for M_{L_1} is in P, so it can make at most polynomial queries to M_{L_2}. Hence, M_{L_1} can simulate M_{L_2} and solve the membership problem.

Theorem 3 *Every language $L \in \mathtt{NP}$ is Cook-reducible to a sparse language iff $\mathtt{NP} \subseteq \mathtt{P}^{\mathtt{poly}}$.*

This theorem is significant because we strongly believe that no NP -complete language is sparse! So, we believe that $\mathtt{NP} \not\subset \mathtt{P}^{\mathtt{poly}}$.

Since SAT is NP -complete, we simply need to show that SAT is cook-reducible to a sparse language iff $\mathtt{NP} \subseteq \mathtt{P}^{\mathtt{poly}}$.

We will exhibit polynomial-time advice string for all inputs of a given length, to use the power of $\mathtt{P}^{\mathtt{poly}}$.

Proof. **(Forward) SAT Cook-reducible to a sparse language $L \implies SAT \in \mathtt{P}^{\mathtt{poly}}$**

There is a polynomial-time machine M which can solve SAT given oracle access to sparse language L.

We want to show that SAT is in $P^p oly$.

Let M run in time $p(n)$ on inputs of length n

The advice string $a(n)$ we want to give is the oracle behaviour on sparse language L. Since the machine M can ask for string of length at most $p(n)$.

Since the language is sparse, the set of all strings of a given length in L is polynomial. So, $a(n) = concat(\{w \in L \mid |w| \leq p(n)\})$ where *concat* concatenates all the strings. $a(n)$ will be polynomial in length since the length of each string w is bounded by $p(n)$. Let $sparse(n)$ be the polynomial that controls the sparsity of L for any string n. That is, for any length i, the language L contains at most $sparse(i)$ strings.

The total number of strings in $a(n)$ will be $N = \sum_{i=0}^{p(n)} sparse(i)$, which is a polynomial in n. Hence, $a(n)$ is a legal advice string.

We're done here, we converted oracle access to a sparse language into a polynomial advice string.

(Backward) $SAT \in \texttt{Ppoly} \implies$ **SAT Cook-reducible to a sparse language** L

We are given a machine M_{sat} which seeks advice $a(n) : \mathbb{N} \to \{0,1\}^*$. The machine M_{sat} runs for polynomial time $p_sat(n)$.

We need to construct a sparse language L_{sparse}, such that given oracle access to L_{sparse}, we can solve SAT using a new machine M'.

Consider all strings that are queried by M_{sat} to M_{poly}. For an input of length n, the machine M_sat can query a $p_{sat}(n)$ times at maximum. Hence, we the language consisting of the subset of a that is sampled by M_{sat} is a sparse language. Given access to this language, we can substitute the function a with the sparse language which contains all advice accessed from a.

Chapter 3

Gaps in space and time

We wish to study what is not computable given some resource. If there resource is time, we want to understand what can be solved in $t(n)$ but not in smaller than $t(n)$ — in the sense of $o(t(n))$.

We can try to construct a hierarchy of problems that can be solved given increasing time.

$$f(n) \in o(g(n)) \equiv \lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$$
$$f(n) \in O(g(n)) \equiv \lim_{n\to\infty} \frac{f(n)}{g(n)} \in O(1)$$

3.1 Space Hierarchy

A function $f : \mathbb{N} \to \mathbb{N}$ is said to be **space constructible** if there exists a turing machine that on input 1^n, it computes $f(n)$ using space $O(f(n))$. So the output can be $1^{f(n)}$ say, since that uses space $O(f(n))$.

Most common functions such as polynomials, exponentials, and logarithms are all space constructible.

Theorem 4 *Let f be a space-constructible function. There exists a language L which can be decided in $O(f(n))$ space, but not in $o(f(n))$ space.*

Proof. The proof is to **construct** a language which can be decided on $O(f(n))$ space, but not in $o(f(n))$ space. Such a language tends to be artificial due to the construction having to work *for all* f.

We need two properties for this language L we create:

- It is **not decidable** in $o(f(n))$ space.
- It **is** decidable in $O(f(n))$ space.

We will use diagonalization to show an construct an L that **cannot be decided** in $o(f(n))$ space. List each TM that runs in $o(f(n))$ space. This collection of all TMs (viewed as strings) is written as:

$$ALLTM = \cup_{i=0}^{\infty}\{0,1\}^i$$

We will define a language L which cannot be decided by **any** TM on the above list.

We will create a matrix of the form $DECIDE(i,j) = M_i(\langle M_j \rangle)$. That is, we feed M_i the string of M_j.($\langle M_j \rangle$ interprets the machine M_j as a string).

Now, create a language L:

$$L \equiv \{M \mid M(\langle M \rangle) = 0\}$$

Note that L is **not decidable** in $o(f(n))$ space. Proof by contradtiction: Assume such a machine M_c (c for contradiction) exists. We now ask if $\langle M_c \rangle \in L$?

- If $\langle M_c \rangle \in L$, then $M_c(\langle M_c \rangle) = 0$ (by the definiton of L). But since M_c **decides** L, $M_c(\langle M_c \rangle) = 0 \implies \langle M_c \rangle \notin L$. **Contradiction**.
- On the other hand, say that $\langle M_c \rangle \notin L$, then $M_c(\langle M_c \rangle) = 1$ (by the definition of L). But since M_c **decides** L, $M_c(\langle M_c \rangle) = 1 \implies \langle M_c \rangle \in L$. **Contradiction**.

We now move to show that L **can be decided** in $O(f(n))$ space. Consider a machine `INTERPRET` that does this:

```
def INTERPRET(w):
    Mw = convert_to_TM(w)

    # Naive solution: Try to run Mw, see what happens.
    # flag = Mw.run(w)

    # Problem 1: How do we know it runs in o(f(n)) space?
    # flag = Mw.run_with_bounded_space(w, space_bound=f(n))

    # Problem 2: How do we know that Mw halts?
    # Count the size of the config. space, and reject if Mw
    # takes more steps than the configuration space size.
    flag = Mw.run_wth_bounded_space_and_steps(w, space_bound=f(n),
                                              steps_bound=Mw.config_space_size())


    return !flag
```

For more details, read Sipser chapter 9

Corollary 1 *For two functions $f1, f2 : \mathbb{N} \to \mathbb{N}$, if $f1 \in o(f2)$, then $DPSACE\,(f1) \subsetneq DPSACE\,(f2)$. (**Sid note: we do not need the condition that $f1 \neq f2$ thanks to the fact that in $o(n)$, the limit tends to** 0)*

3.2 Time Hierarchy

Theorem 5 *Let f be a time-constructible function. There exists a language L which can be decided in $O(f(n))$ time, but not in $o\left(\frac{f(n)}{\log(f(n))}\right)$ time.*

Proof. Proof is the same as that of space hierarchy (roughly).

We get the log factor for us to simulate a $f(n)$ time turing machine. We do not know how to perform the simulation with constant overhead.

Corollary 2 $P \subsetneq EXPTIME$

3.3 Polynomial Hierarchy

One interesting thing to study is the power of oracles (non-uniform computations). One can try to study the nature of languages, given oracle access.

Definition 1 *Let M be a turing machine, A be a language. **The language** $L(M^A)$ is the set of strings accepted by the machine M with oracle access to A.*

We can deneralize this by giving access to a *class of languages*!

Definition 2 *Let M be a turing machine, C be a class of languages. **The language** $L(M^C)$ is the set of strings accepted by the machine M with oracle access to any language in C.*

$$M^C = \{L(M^A) \mid A \in C\}$$

Definition 3 *Let C_1, C_2 be classes of languages. **The language** $L(C_1^{C_2})$ is the set of strings accepted by some machine in C_1 given oracle access to some machine in C_2.*

$$C_1^{C_2} = \{L(M_1^{M_2}) \mid M_1 \in C_1, M_2 \in C_2\}$$

We will use M^ϕ to denote oracle access to an "empty" oracle. Hence, $M^\phi \sim M$.

An example would be $\texttt{co-NP} \subset \texttt{P}^{\texttt{NP}}$, because the `P` oracle can flip the answer of the `NP` oracle.

Chapter 4

Polynomial Hierarchy

Chapter 5

Probabilistic proofs

5.1 IP — interactive proofs

Definition 4 *Completeness: For every true assertion, there is a valid proof.*

Definition 5 *Soundness: For every false assertion, no valid proof exists.*

A good proof system must also be such that the verifier is efficient (that is, polynomial time).

If we ask that a proof system must be sound and complete, there is no scope for error! Further, it is not clear if the verifier and the prover can "talk" to each other. If we choose to allow interactions, what are the implications?

We relax the assumptions this way — Relaxed compleness states that for every true assertion, there is a proof strategy that will convince the verifier with probability at least $> \frac{2}{3}$. Similarly, relaxed soundness states that for every false assertion, every proof strategy fails to convinve the verifier with probability at least $> \frac{2}{3}$.

The formalization is as follows:

Definition 6 *Interactive proof systems*

- *An interactive proof system for a language L consists of two entities: a prover P and a verifier V. P and V share common input, and work for $R \in \mathbb{N}$ rounds.*
- *In each round, the prover can send the verifier a message that is polynomial in the length of the input.*
- *The verifier can send a polynomial length reply to the prover.*
- *The verifier is a randomized polynomial time turing machine. Time is measured as a function of the length of the input.*
- ***Completeness**: $\forall x \in L$, there exists a prover strategy so that the verifier accepts with probability $> \frac{2}{3}$.*
- ***Soundness**: $\forall x \notin L$, any prover strategy will lead the verifier to accept with probability $< \frac{1}{3}$.*

Note that the power of the prover in unspecified in this definition — we are implicitly saying that finding a proof is generally much harder than verifying a proof. Hence, the prover has no real bounds on the power, while the verifier does.

We also have the value $R \in \mathbb{N}$, which lets us setup the number of rounds. This is a knob we can twiddle, that allows us to change the hardness of the problem.

Definition 7 *The IP hiearchy: Let $r : \mathbb{N} \to \mathbb{N}$ be the "number of rounds" function. Define $IP(r)$ to be the set of languages such that there exists an interactive proof system using at most $r(|x|)$ rounds on input x.*

For a class of functions $R \subset \{\mathbb{N} \to \mathbb{N}\}$, we can then define $\texttt{IP}(R) = \cup_{r \in R} \texttt{IP}(r)$.

Note that NP $\subset$ IP. Also, the number of rounds cannot be more than polynomial — the verifier is poly bounded in time, so the verifier cannot work more than poly rounds. So, we denote $\texttt{IP} \equiv \texttt{IP}(O(poly(n))$.

Both **randomness** and **interaction** are essential to the definition.

When randomness is removed but only interaction is present, this will be like NP . The prover can arrive by itself the set of messages the verifier would send to the prover.

When interaction is removed but randomness is remained, the verifier is similar to that of NP , but the verifier can now be **probabilistic**. This class of languages is likely beyond NP .

5.2 Graph non-isomorphism (GNI)

Two graphs G, H are isomorphic (denoted $G \sim H$), iff there exists a bijection such that $\forall x, y \in V_1, (x, y) \in E_1 \implies (f(x), f(y)) \in E_2$.

Using this, we define GNI , the problem of checking if two graphs are not isomorphic:

$$\texttt{GNI} \equiv \{\langle G, G' \rangle \mid G \nsim G'\}$$

Graph isomoprhism is in NP since the witness will just be the bijection. Hence, GNI is in co-NP, and it is not known whether GNI is in NP .

In an interactive proof system for GNI , the verifier asks the prover to distinguish between isomorphic graphs.

- G_1, G_2 are given to both prover, verifier.
- The verifier chooses a random $r \in \{1, 2\}$ uniformly at random.
- The verifier picks a random permutation π of the set $\{1, 2, \ldots, |V(G_1)|\}$
- the verifier constructs the graph H as the permutation of G_r under π. The graph H is sent to the prover. That is, $H \equiv \pi(G_r)$.
- the prover P replies with $r' \in \{12\}$. The reply r' is 1 if H is isomorphic to G_1, and 2 otherwise.
- The verifier accepts if $r = r'$.

Note that $H \sim G_r$. Now if $G_r \sim G_{other}$, then $H \sim G_r \sim G_{other}$, and so the prover has to literally guess between G_r and G_{other}, and at best it can simply guess. (Even though the prover has *unbounded computation*, it is unable to distinguish between G_r and G_{other}). In two rounds, the probability of the guesses of the prover being right is $\frac{1}{2}^2 = \frac{1}{4}$, which fulfils our soundness guarantee ($\frac{1}{4} < \frac{1}{3}$).

On the other hand if $G_r \nsim G_{other}$, then if the prover knows how to solve `GNI` , it can check between H, G_r, and G_{other} to consistently report G_r. In this case, the prover will *always* be correct, so this will pass compleness (since $1 > \frac{2}{3}$).

This is very interesting, because the verifier **does not know** whether $G_1 \sim_? G_2$. The verifier tries to engage with the prover, to understand whether $G_1 \sim_? G_2$.

Theorem 6 $GNI \subset IP(2)$

Theorem 7 $co\text{-}NP \subset IP$

Theorem 8 $IP = PSPACE$

Chapter 6

Parallel Computing

Moore's law, blocking factors:

- Memory wall: memory latency was higher than compute.
- Power wall: Power leakage.
- ILP wall: ILP via branch prediction, out-of-order-execution, and speculative execution. Diminishing returns from instruction-level parallelism.

Interesting questions one can ask:

- How do we analyze parallel algorithms?
- Can every sequential algorithm be parallelized?
- What are the complexity classes related to parllel computning?
- Can sequential programs be automatically converted to parallel programs?

Concurrent data structures, course: Professor Govindarajulu.

6.1 `PRAM` model

Global shared memory, shared by n processors. Each processor has individual bidirectional buses into the memory.

Also, note that we have *random access* into the memory, which is different from a turing machine, which only offers sequential access.

(Sid question: what is a problem that can be solved efficiently given random access and not with sequential access?)

Access to shared memory costs the same as one unit of computation.

Different flavours provide different semantics to concurrent access of shared memory (EREW, CREW, CRCW).

- EREW - Exclusive Read, Exclusive Write. No scope for memory contention, so algorithm design is tough.

- CREW - Concurrent Read, Exclusive Write. Allow processors to read simeltaneously, writes are still exclusive. Is practically feasible.
- CRCW - Concurrent Read, Concurrent Write Processors can read/write simeltaneously. So here, we need to specify semantics of concurrent writes!

Flavours of concurrent write semantics:

- COMMON: Concurrent write is allowed as long as all the values being attempted are equal. Eg: boolean OR of n bits. Each processor p_i will read $a[i]$. if $a[i] = 1$, then p_i tries to write 1. it doesn't matter how many processors try to write 1, if any bit is 1, then the output will be 1. We need to make sure the cell is initialized to 0, so that if all bits are 0, the answer is 0.
- ARBITRARY: In the case of a concurrent write, *someone* wins and its write takes effect.
- PRIORITY: Assumes that processors have numbers that can be used to decide which write succeeds.

6.2 Matrix multiplication

- Recursively block the matrices.
- Multiply strips (cannon's algorithm).

6.2.1 Prefix computations

Given an array A of n elements and an associative operator $\circ$, we want to compute $P(i) = \circ_{k\in[0\ldots n]}A[k]$. $P(0) = A(0), P(1) = A(0) \circ A(1) \ldots$.

We can use the naive approach:

```
def scan(A, op):
    out[0] = A[0]
    for i in range(1, length(A)):
        out[i] = op(A[i], out[i - 1])
```

We have linear RAW dependences: $out[i] \to out[i-1]$.

So, we crete a complete binary tree with processors at the internal nodes. Input is at the leaf node. Each node performs the $\circ$ of its left and right subtree.

```
def sumfast(A, op, l, r):
    if (l == r):
        return A[l]
    else:
        mid = (l + r) / 2
        return op(scanfast(A, op, l, mid), scanfast(A, op, mid, r));

def sum(A, op):
    return sum(A, op, 0, length(A))
```

Note that this will not give us *prefix sums*. We can finish in $\log(n)$ time given 2^n processors.

For a *prefix sum*, we need a combination of upward and downward traversal. First send data from bottom to top. Next, send down data from top to bottom of the prefix sums towards the leaves.

Analysis of prefix computation

- Step 1 can use $\frac{n}{2}$ processors in parallel, each using 1 unit of time.
- Step 2 is a recursive calls and takes $T(\frac{n}{2})$ time.
- Step 3 uses $n$processors each of which take 1 unit of time.

Work done by the algorithm: $W(n) = W(n/2) + O(n)$ ($O(n)$ for the first and third step). $W(n) = O(n)$ is the solution.

Optimal parallel algorithm

A parallel algorithm that does the same amount of work as the best known sequential algorithm is called an *optimal algorithm*.

This makes sense, because if we set `num processors` $= 1$, we want the asymptotics to match the sequential algorithm.

Chapter 7

Design models of parallel algorithms

7.1 Partitioning

This is similar to divide-and-conquer, but we don't need to *combine* solutions! We can treat problems independently and solve it in parallel. Examples are parallel merging and searching.

We generate subproblems that are independent of each other. Example is quicksort. Once we partition the array into two subarrays, we sort the subarrays recursively.

7.1.1 Merging in parallel by partitioning

Two sorted arrays A and B are to be merged into an array C.

suboptimal algorithm — Time: $O(\log n)$**, work:** $O(n \log n)$

We define a function $Rank(x_0, X) = |\{x < x_0 \mid x \in X\}|$. Note that the position of x_0 in $sorted(X)$ is equal to $Rank(x_0, X)$. **Claim:** $Rank(x, C) = Rank(x, A) + Rank(x, B)$.

For $x \in A$, $Rank(x, A)$ is immediately available (since A is sorted). We need to find $Rank(x, B)$, but we can find this using binary search through B.

Time for each binary search is $O(\log n)$. Total time for merging is $O(\log n)$, since we are doing each binary search in parallel — we just need to read the array B, no need to update. The total work is $O(n \log n)$, since we are performing $O(\log n)$ work for n elements.

Note that this is **non optimal**. The sequential algorithm has a time complexity of $O(n)$.

We are going to try and reduce the work to $O(n)$.

Merging, take 2, optimal — time: $O(\log n)$**, work:** $O(n)$

General technique is to solve a smaller problem in parallel, and then extend the solution to the entire problem!

- The problem size to be solved is guided by the factor of non-optimality in the current algorithm. We need to reduce the total work to $O(n)$.

  For input size n, we do $O(n \log n)$ work. So, for input size $n/\log n$, we do $O(n/\log n \times \log(n/\log n) \sim O(n) + O(\log(\log(n)) \sim O(n)$.

- We pick every $\log n$th element of A. We merge the selected elements of A and B. However, we still perform binary search on the entireity of B.
- Pick elements $A[\log n], A[2\log n], \ldots, A[n - \log n], A[n]$, and rank the, in B (ie, find their corresponding positions in B.)
- Define $[B_{r(i)}, \ldots, B_{r(i+1)}] \equiv$ portion of B between $A[r \log i]$, $A[(r+1)\log i]$ in B.

```
A = (5) 6 9 12 (15) 18 19 (21) 23 26
B = 1 4 (..5..) 7 8 10 11 12 (..15..) 16 17 20 (..21..) 22

In the output array, we can merge
the array of B between the (..) elements of A
```

  The problem is that the size of $\log n$ per chunk in A does not mean that the size is $\log n$ in B.

```
A = (5) 6 9 12 (15) ... (...) ...
B = 6 6 6 6 6 6 6 ... 6

In this case, the entireity of B is between [5, 15]
```

  So, if we can somehow control the size of B, so, we can perform binary search in $O(\log n)$, with $n/\log n$ processors.

  We then need to perform the merge with $O(\log n)$, **under certain conditions**. There are again $n/\log n$ such merges.

  The work is $O(n)$.

  So now, the only thing we need to control is the size of partitions of B.

- If $[B_{r(i)}, \ldots, B_{r(i+1)}]$ is too large, then we can pick $\log n$ items of this section, and we can rank them in A! Each piece in A will be smaller than $\log n$, since the partition of A was already $\log n$.
- we can merge two sorted arrays of size n in time $O(\log n)$ with work $O(n)$. This algorithm works in `CREW`. We can improve this further, we will see this later.

7.1.2 Searching faster — time: $O(1)$, work: $O(\sqrt{n})$

Each binary search takes $O(\log n)$ time, and we have $O(n/\log n)$ subproblems, each of size $O(\log n)$.

Can we make search faster?

- Consider a sorted array A with n elements. We want to search for an element x. Given p processors, we can search at the indeces $1, n/p, 2n/p, \ldots, n$.
- Record the result of each comparison as 1 or 0. $cmp[i] = 1 \equiv A[i] < x$, $cmp[i] = 0 \equiv A[i] \geq x$. More succinctly, `cmp = map (\a -> a < x) A`.

- *cmp* will either have all 0s, all 1s, or a shift from 1s to 0s.
- If we have a shift from 1s to 0s, we know that x is likely in the n/p segment corresponding to the shift from 1 to 0.
- So now, we can recursively search that small segment.
- $T(n) = T(n/p) + O(p)$. ($O(p)$ since *cmp* has length p). Hence, $T(n) = T(n/p) + O(1)$. This gives us $O(\log n)$ when $p = 1$ (make sure this is correct, there is some **off by one here**.
- When $p = O(\sqrt{n})$, the time taken will be $O(\log n / \log(\sqrt{n})) = O(1)$ This looks useless from a work point of view, but we want to see what this is good for!

7.1.3 From parallel search to merge — time: $O(\log \log n)$, work: $O(???)$

- We have two sorted arrays A and B, which we want to merge.
- We want to rank some elements of A to create paritions of B.
- Let us take $\sqrt{n}$ elements of A in B.
- We have n processors, so each search can use $n/\sqrt{n} = \sqrt{n}$ processors.
- each search now finishes in $O(1)$ time.
- the problem is that the partitions of A are much larger now (they are $\sqrt{n}$ large).
- we have a $\sqrt{n}$ sized piece of A, and we have a size of B that is of size (?). Note that for each piece of A, we now choose to allocate $\sqrt{n}$ processors.
- So, we pick $n^{\frac{1}{4}}$ elements of A in B, each of which uses $n^{\frac{1}{4}}$ processors. Size of each piece is now $n^{\frac{1}{4}}$.
- So, we pick $n^{\frac{1}{8}}$ elements of A in B, each of which uses $n^{\frac{1}{8}}$ processors. Size of each piece is now $n^{\frac{1}{8}}$.
- We reduce the sequence $n \to \sqrt{n} \to n^{\frac{1}{4}} \to n^{\frac{1}{8}} \cdots \to O(1)$. This can be done in $\log \log n$ steps!

Chapter 8

Parallel algorithms, part 2

8.1 Pointer jumping

Pointer jumping is the technique of updating a successor with the successor's successor. As this is repeated, the sucessor gets closer to the root node. The distance between a node and its successor doubles in each round trip.

```
# F := Forest consisting of rooted, directed trees. F is specified using
# an array P

# P[i] := P[i] = j iff (i, j) is an edge in F. That is, j is a parent
# of i.

# P must contain self-loops at *each of the roots*. Each arc is
# specified by (i, P[i])

# output: a list S, containing the root of i at S[i]
def find_roots(P):
    for i in parallel([1, n]):
        S[i] = P[i]

        while S[i] != S[S[i]]:
            S[i] = S[S[i]]

    return S
```

8.2 List Ranking

We have a list L of n nodes. $S[i]$ contains a pointer to the node *following* node i on L, for $1 \leq i \leq n$. We assume that $S(i) = 0$ when i is the end of the list. The *List-ranking problem* is to determine the distance of each node i from the end of the list.

8.2.1 non-optimal list ranking using pointer jumping

```
def listrank(S):
    for i in parallel([1, n]):
        S[i] =  R[i] == 0 ? 0: 1


    for i in parallel([1, n]):
        Q[i] = S[i]
        while Q[i] != 0 && Q[Q[i]] != 0:
            R[i] = R[i] + R[Q[i]]
            Q[i] = Q[Q[i]]
```

this takes time $O(\log n)$, using $O(n \log n)$ operations.

8.2.2 Making our algorithm better

We want to make our algorithm better, we have a work complexity of $O(\log n)$ which we are trying to eliminate.

There are also some implementation issues. In the PRAM model, syncrhonous execution means that all n processors execute each step in parallel. So, we can have inconsistent results!

How do we pick a list of size $n/\log n$? Our input is in the form of an array of successor elements. So, we can't take equi-distant parts of the array, since it won't be a valid sub-list anymore.

What we can do is to pick *independent nodes*. Formally, we want to remove an independent set: vertices that share no edge amongst them.

```
1 -> (8) -> 5 -> 11 -> (2) -> 6 -> (10) -> 4 -> 3 -> (7) -> 12 -> 9
on removal:
1 -> 5 -> 11 -> 6 -> 4 -> 3 -> 12 -> 9
```

We can remove `8, 2, 10, 7` in parallel.

We want to go to a subset of size $n/\log n$, but by removing independent nodes, we can go smallest to $n/2$.

```
a -> (b) -> c -> (d) -> e -> (f) -> ...
```

There are no other elements in the above chain we can add to the independent set. So, we will need to repeat our process to reach $n/\log n$.

8.3 Detour: Independent sets

In a graph $G = (V, E)$, a subset of nodes $U \subseteq V$ is called an *independent set* if:

$$U \text{ is an independent set of G} \equiv \forall (u_1, u_2) \in U, u_1 \neq u_2 \implies (u_1, u_2) \notin E$$

.

Linked lists, when viewed as graphs, have large independent sets.

8.3.1 Technique: Symmetry breaking

The idea is to look at a symmetric setting, and then induce differences between them. Independent sets are symmetric, because given two nodes that are neighbours, they're both eligible to be in the independent set (modulo other obstructions). This algorithm is applicable for graph coloring.

Usually, this technique requires randomization. However, there are special cases where fast, deterministic symmetry breaking is possible. Linked lists and directed cyclic graphs are examples where this is possible.

We first construct a symmetry-breaking based graph coloring solution, which is then used to find independent sets.

8.3.2 Coloring by Symmetry breaking

Considered a directed cycle of n nodes $0 \dots n-1$.

Assume we have 8 nodes, which are labeled using 3 bits. We may not have consecutive numbering of our nodes, so we assume that our nodes are randomly numbered, from 0 to 7 (3 bits).

- Initially, treat each number as a color for the vertex.
- We can reduce the number of colors to $\log n$ in one step:
  - Compare color with the parent. $Newcolor(u) = 2k + color(u)[k]$.
  - k is the index of the first bit position from LSB where $color(u)$ and $color(parent(u))$ differ.
  - So, $color(u)[k]$ is indexing the k-th bit of $color(u)$ starting from LSB.
  - note that $0 \leq k \leq \log n - 1$.
  - such a k will always exist, since we are guaranteed some unique labelling of the vertices when we start this process.

```
This table may not be fully accurate, re-check:

u   | v   | new color (mostly 2 bits)
110 | 000 | 11 (k = 1)
000 | 100 | 100 (k = 2)
100 | 111 | 00 (k = 0)
010 | 001 | 00 (k = 0)
001 | 011 | 10 (k = 1)
011 | 101 | 11 (k = 1)
111 | 010 | 01 (k = 0)
101 | 110 | 01 (k = 0)
```

Correctness proof

Proof by contradiction. Suppose we have an edge (u, v), where $newcolor(u) = newcolor(v)$. Let $newcolor(u) = 2k + color(u)[k]$, and $newcolor(v) = 2r + color(v)[r]$.

If $newcolor(u) = newcolor(v)$, then $2k + color(u)[k] = 2r + color(v)[r]$. Rearranging, we get that $2(r-k) = color(u)[k] - color(v)[k]$.

If $k = r$, then we get that $color(u)[k] = color(v)[k]$. But this cannot happen, because by definition, k is the bit where u, v first differ!

If $k \neq r$, then we get that $2(r-k) = color(u)[k] - color(v)[k]$. By comparing magnitudes, we see that $\left|color(u)[k] - color(v)[k]\right| \leq 1$ (since we're subtracting bit values), while $\left|2(r-k)\right| \geq 2$. This can't happen either for two equal values!

Analysing number of new colors

In one iteration, we can reduce the number of colors from n to $2\log n$. For the new colors, we only need $1 + ceil(\log\log n)$ bits.

Can we repeat this technique? Yes, we can. This technique reduces number of colors from t to $1 + ceil(\log t)$. At some point, $t < 1 + ceil(\log t)$, at which point we will be forced to stop.

This stopping point happens at $t = 3$. So, we repeat until only 8 colors are being used.

The total time is the solution to the recurrence $T(n) = T(\log n) + 1$. We define the function that solves the recurrence as $\log^* n$.

$$\log^* n = i \equiv \log(\log(\ldots i \text{ times} \ldots (n))) = 1$$

Reducing from 8 to 3 colors

for i in [8..3], If node u is colored i, then choose a color among $\{1, 2, 3\}$ that is not the same as its neighbours.

```
# color: map (vertex -> color)
# V: vertex set
for c in range(8, 3):
    for v in V:
        if color[v] == c:
            # we will always have one number here, since we have three
            # colors, and we are only removing two colors
            newcolor[v] = rand ({1, 2, 3} - color[pred(v)] - color[succ(v)])
    newcolor = color
```

This is always possible.

8.3.3 Finding Indepenent sets using the coloring

For bounded degree graphs colored with $O(1)$ colors, a coloring is equivalent to finding a large independent set.

Iterate on each color and count the number of nodes with a given color. Pick the subset of like colored nodes of the largest size. It is clearly an independent set, and has size of at least some fraction of n.

8.3.4 Algorithm outline

```
def rank(L):
    L1 = L

    for r in [2, R]:
        color the list with 3 colors
        pick an independent set U_i of nodes of size >= n /3
        L_i = remove nodes in U_i from L_{i - 1}

    Rank the List L_r using poiner jumping


    for i in [r, 1]:
        reinsert the nodes in U_i into L_i
```

We are removing $n/3$ nodes in each iteration, we want to stop at $n/\log n$ nodes. We need $O(\log\log n)$ iterations.

8.3.5 total time taken

Each iteration is $O(\log^* n)$. At $O(\log\log n)$ iterations, this takes $O(\log^* n \log\log n)$ time.

To rank the remaining list, we take $O(\log n)$ time.

To reintroduce the removed elements, we take $r = O(\log\log n)$ iterations, $O(\log\log n)$ time.

8.3.6 Slowing down re-introduction to make this optimal

We can reintroduce slower.

we can use only $n/\log n$ processor

8.3.7 Slowing down independent set

8.4 Back to list ranking

- Anderson-Miller is in JaJa's book
- Hellman-JaJa is another popular approach (read the paper)

Chapter 9

Tree processing

9.1 Traversal via an Euler tour

Definition 8 *an **Euler tour** is a cycle of a graph that includes every edge of the graph exactly once.*

Lemma 2 *A directed graph G **has an Euler tour** iff for every vertex,its in-degree equals its out-degree.*

For a tree $T = (V, E)$, to define an euler tour, we make it a directed graph. $T_e = (V_e, E_e)$, where $V_e = V$, and $E_e = \cup_{(u,v)\in V}\{(u, v), (v, u)\}$ That is, each (u, v) in E creates two edges (u, v), and (v, u) in E_e. T_e will have an Euler tour.

We have to define a successor function $s : E_e \to E_e$. Here, the successor for an edge. For a node u in T_e, order its **neighbours (both incoming and outgoing)** $v_1, v_2, \ldots v_d$. This can be done **independently at each node**. For $e = (v_i, u)$, set $s(e) = (u, v_{i+1 \bmod d})$. This choice of s is valid since we always have both edges (x, y) and (y, x), and we are therefore assured that (v_i, u) will be an incoming edge, and $(u, v_{i+1}$ will be an outgoing edge. Also, compute $i + 1$ modulo d, so that we eventually cycle.

TODO: relabel vertices to $[0..(d-1)]$ **so that modulo works properly TODO: add example**

Theorem 9 *s actually constructs a tour.*

Proof. Induction on number of vertices. If $n = 1$, obviously true. If $n = 2$, at most one edge present. We will go along the edge and come back, which is a valid tour.

- Let the tour be well defined for $n = k$. We will prove it for $n = k + 1$.
- Every tree has at least one leaf, call it l. Create a tree $T' = T/\{l\}$.
- Let u be a neighbour of l in T.
- Let $N(u) = \{v_0, v_1, \ldots v_i = l, v_{i+1}, \ldots v_d\}$.
- Set $s_{new}(u, v) \equiv (v, u)$. Set $s_{new}(v_{i-1}, u) \equiv (u, v)$.
- For all other vertices, $s_{new}(e) = s(e)$.

9.2 Using euler tours for traversal

Operations on a tree such a rooting, preorder, and postorder traversal can be converted to routines on an Euler tour.

9.2.1 Rooting a tree

Designate a node in a tree as the root. All edges in the tree are directed towards (or away) from the root.

- let $\{v_1, v_2, \ldots v_d\}$ be the neighbours of root node r.
- we mark the final edge of the tour as `NIL`, so we get an Euler path, and not an Euler tour.
- the edge (r, v_i) appears before (v_i, r).
- so the edge $parent \to child$ appears before $child \to parent$
- So, if uv precedes vu, then set $u = parent(v)$. Orient the edge uv as $v \to u$ (that is, $child \to parent$), since we want all edges towards the root.

9.2.2 Preorder traversal

We have a rooted tree with r as the root. In a preorder traversal, a node is listed before any of the nodes in its subtrees.

In an Euler tour, nodes in a subtree are visited by entering subtrees, and the exiting towards the parent.

If we can track the first occurence of a node in an euler path, this will tell us the preorder traversal. Note that edges in the euler tour occur first as $parent \to child$, and later as $child \to parent$. So, we can look at the sequence of edges in the euler tour, and find the preorder numbering.

9.2.3 Expression tree evauation of binary trees

Tree may not be balanced.

We use the `RAKE` technique to evaluate subexpressions. We rake the leaves from the expression tree — we remove the leaf node and its parent.

- $T = (V, E)$ is a tree rooted at root node r. $p : E \to E$ is the parent function.
- One step of the rake operation at a leaf l with $p(l) \neq r$ involves:
  - Remove node l, $p(l)$ from the tree
  - Make the sublings of l as the child of $p(p(l))$. That is, graft the siblings of l to the grandparent of l.

Why is this a good technique? Can this be applied in parallel to several leaf nodes? Yes, it can be applied to leaf nodes that don't share the same parent. In general, there is a richness of leaf nodes in a tree, since there are only $n - 1$ edges.

Each application of rake at all leaves reduces the number of leaves by half. Each application of `RAKE` is $O(1)$. So, total time is $O(\log n)$.

```
def shrinkTree(R):
    compute labels for leaf nodes, store in array A (exclude leftmost
    and rightmost nodes in this A)

    for _ in range(k):
        apply rake operation to all odd numbered leaves that are
        the *left* children of their parent

        apply rake operation to all odd numbered leaves that are
        the *right* children of their parent

        update A to be the remaining even leaves
```

Applying `Rake` means that we can process more than one leaf node at the same time.

Fo expression evaluation, this may mean that an internal node with only one operand gets raked.

```
          + g(u)

    + p(u)

Y     X (u)


--After raking--

   + g(u)
Y
```

- Transfer the impact of applying the operaot at p(u) to the sibling of u
- $R_u = a_u X_u + b_u$
- X_u is the result of the subexpression at node u – $X_u = f(left, right)$
- adjust a_u and b_u during any rake operation appropriately
- Initially, at each leaf node, $a_u = 1, b_u = 0$.

```
          + g(u)

    + p(u)
    X_w
    a_w
    b_w

v        (u) 5, 1, 0
```

```
X_v,
a_v,
b_v

--After raking--

     + g(u)
v
X_v'
a_v'
b_v'
```

- Before removing $p(u)$, the contribution of $p(u)$ to $g(u)$ will be $X_w a_w + b_w$.
- we want what $p(u)$ used to calculate to be what v calculates after.
- $X_w = (X_u a_u + b_u) + (X_v a_v + b_v) = (X_v a_v) + (X_u a_u + b_u + b_v)$
- What $p(u)$ used to calculate is: $a_w X_w + b_w = a_w(a_v X_v + a_u x_u + b_u + b_v) + b_w = a_w a_v x_v + a_w(a_u X_u + b_u + b_v) + b_w$
- what $p(v)$ should be: $a'_v = a_w a_v$, $b'_v = a_w(\dots)$

For other operators, proceed in a similar fashion (**TODO: do this and send to kiko, he seems interested!**)

Chapter 10

Tree processing, mach 2

Chapter 11

Parallel Graph algorithms

We now move to parallel graph algorithms. We will view recent work on 1-connectivity and 2-connectivity.

A graph is 1-connected if every pair of vertices has a path between them.

In a sequential model, DFS/BFS can be used. In the parallel setting, we know that DFS cannot be parallelized. BFS can be, but it is inefficient to do so (We will see this later). So, we need new approaches to this problem.

A connected component is a subset of vertices V_i such that every pair of vertices in V_i have a path between them.

The algorithm we study has some resemblance with the union-find algorithm.

11.1 The algorithm for 1-connectivity

The algorithm is by **Chandra, Sarwate, Hirschberg**.

11.1.1 Intuition

- Consider an initial set of rooted trees where each tree contains a single vertex.
- Eventually, each rooted tree will correspond to a connected component.
- Two trees can be combined into a bigger tree if these trees contain vertices u and v which belong to different trees, and the edge $(u, v) \in E(G)$. The next iteration proceeds with trees merged from the previous iterations.
- The parallelism is in merging the trees

Instead of calling it a tree, we call it a *super-vertex*.

We define the *graph for an iteration* as the graph of super-vertices and edges between the super-vertices.

- $G_0 = G$
- G_i is the graph with super-vertices at iteration i. We construct G_{i+1} from G_i.

Important questions:

- How to represent and arrange the super-vertices?
- How do we build the graph for the next iteration?
- How many iterations do we need?
- What is the time and work complexity of the algorithm?

11.1.2 How to represent the matrix?

We will use an adjaceny matrix to start with. Initially, the matrix is of size $n \times n$ where $n = |V(G)|$.

If the graph at the start of the kth iteration has n_k vertices, then the matrix is of size $n_k \times n_k$. We will refer to this matrix as A_k.

$A_k[u, v] = 0$ means that the super-vertices do not share an edge. $A_k[u, v] = 1$ if u and v do share an edge.

11.1.3 How do we build the graph for the next iteration?

We will make use of *concurrent writes* to create the matrix A_{k+1} from the graph G_k.

In G_k, if there exist two distinct super-vertices u_s and v_s such that a vertex u in u_s and a vertex v in v_s and the edge (u, v) is in $E(G)$.

11.1.4 How do we arrange the super-vertices?

Each vertex of G is given a label ($label : V(G) \to \mathbb{N}$, $label$ is injective) so that if $label(u) = label(v)$, then u and v are part of the same super-vertex.

The common label used for all vertices in the super-vertex will be the label of the smallest numbered vertex in the super-vertex.

- We set this up such that **the root of every tree is the node with the smallest id**.
- As we combine two trees to make a bigger tree, we will make the tree with the lower root id as the parent.
- We use *pointer jumping*f to adjust the labels.

11.1.5 The merging algorithm

We define a function:

$$C : V \to V$$
$$C(v) = min\{label(w) \mid A[v, w] = 1\}$$

In the first iteration, starting with an initial set of n trees, we merge trees as follows:

- C creates a forest of trees on V and with $E = \{(v, C(v)) \mid v \in V(G)\}$.

- C partitions V such that all vertices in the same connected component are in the same partition.
- Each cycle in the forest is either a self-loop or of length 2.
- We now use pointer jumping to make everyone in a tree agree on a representative.

11.1.6 The algorithm

```
A_0 = A
n_0 = n
k = 0
while not done:
        k = k + 1
        for v in V pardo:
                C(v) = min {w | A[k - 1][v][w] == 1}

                Shrink each tree in the forest

        pass
```

11.1.7 Analysis

- Number of iterations: We can show that in each iteration till the ened, the number of super vertices decreases by a factor of two. So, the total number of iterations is $O(\log n)$.
- Time spent in each iteration: In each iteration, we need to do a pointer jumping across the forest, This takes $O(\log n)$time and $O(n)$ work.
- Total time: $O(\log^2 n)$.
- Work: $O(n+m)$ $(n = |V|, m = |E|)$

There are better algorithm that reduce the time to $O(\log n)$ by **Shiloach and Vishkin** — Don't perform aggressive pointer jumping every round, but perform one step of the pointer jumping each round.

11.2 k-Connectivity

Famous result by **Cherian and Thirumella**:

A graph is k-connected iff the subgraph H is k-connected, where H is:

$$
\begin{aligned}
T_1 &= BFS(G) \\
T_2 &= BFS(G/T_1) \\
T_3 &= BFS(G/(T_1 \cup T_2)) \\
T_k &= \dots \\
H &= T_1 \cup T_2 \dots T_k
\end{aligned}
$$

Note that each of the T_i are disjoint, and each T_i may have n edges, so H has only kn vertices. This is drastically better that $|E| = O(n^2)$