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.

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

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

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

1.1.1 Solving $\overline{\texttt{PATH}}$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 $\texttt{P}^{\texttt{poly}}$

$L \in \texttt{P}^{\texttt{poly}}$ 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 $\texttt{P}^{\texttt{poly}}$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 $\texttt{P}^{\texttt{poly}}$contains non-recursive languages

Chapter 2

Advice & Time Hierarchies

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

This class could possibly be bigger than $\mathtt{P}$.

In $\mathtt{NP}$, 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* $\mathtt{P}^{\mathtt{poly}}$

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

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) $SAT \in \mathtt{P^{poly}} \implies$ **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 $\mathtt{P} \subsetneq \mathtt{EXPTIME}$

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 $\mathtt{co\text{-}NP} \subset \mathtt{P}^{\mathtt{NP}}$, 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` *hieararchy: 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 `IP` $(R) = \cup_{r \in R}$ `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 `IP` $\equiv$ `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 $\mathtt{GNI} \subset \mathtt{IP}(2)$

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

Theorem 8 $\mathtt{IP} = \mathtt{PSPACE}$

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.

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 work

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.