12-2020

An Introductory Survey of Computational Space Complexity

Kaylee Noelle Ghent
Trinity University, ghentkng@gmail.com

AN INTRODUCTORY SURVEY OF COMPUTATIONAL SPACE COMPLEXITY

KAYLEE N. GHENT

A DEPARTMENT HONORS THESIS SUBMITTED TO THE DEPARTMENT OF COMPUTER SCIENCE AT TRINITY UNIVERSITY IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR GRADUATION WITH DEPARTMENTAL HONORS

DATE: November 20, 2020

J. Paul Myers, Jr.	Yu Zhang
THESIS ADVISOR	DEPARTMENT CHAIR

Michael Soto, AVPAA

Student Agreement

I grant Trinity University (“Institution”), my academic department (“Department”), and the Texas Digital Library (“TDL”) the non-exclusive rights to copy, display, perform, distribute and publish the content I submit to this repository (hereafter called “Work”) and to make the Work available in any format in perpetuity as part of a TDL, digital preservation program, Institution or department repository communication or distribution effort.

I understand that once the Work is submitted, a bibliographic citation to the Work can remain visible in perpetuity, even if the Work is updated or removed.

I understand that the Work’s copyright owner(s) will continue to own copyright outside these non-exclusive granted rights.

I warrant that:

1) I am the copyright owner of the Work, or
2) I am one of the copyright owners and have permission from the other owners to submit the Work, or
3) My Institution or Department is the copyright owner and I have permission to submit the Work, or
4) Another party is the copyright owner and I have permission to submit the Work.

Based on this, I further warrant to my knowledge:

1) The Work does not infringe any copyright, patent, or trade secrets of any third party,
2) The Work does not contain any libelous matter, nor invade the privacy of any person or third party, and
3) That no right in the Work has been sold, mortgaged, or otherwise disposed of, and is free from all claims.

I agree to hold TDL, DPN, Institution, Department, and their agents harmless for any liability arising from any breach of the above warranties or any claim of intellectual property infringement arising from the exercise of these non-exclusive granted rights.”

I choose the following option for sharing my thesis (required):

[X] Open Access (full-text discoverable via search engines)
[ ] Restricted to campus viewing only (allow access only on the Trinity University campus via digitalcommons.trinity.edu)

Abstract

Using the Understanding by Design pedagogical methodology, this thesis aims to combine, clarify, and contextualize introductory ideas about computational space complexity and package them in an instructional unit. The unit is composed primarily of a Unit Template, series of Lessons, and Performance Assessments. It is intended to present content acknowledged as valuable by ACM that is often missing from undergraduate computer science curricula at peer educational institutions to Trinity University. The unit covers ideas such as the space hierarchy, computational time / space tradeoffs, and completeness, and is designed to promote understanding and inquiry of and beyond its subject matter.

Introduction

The topic of space complexity is almost never a part of required coursework for Computer Science majors in the undergraduate setting. When it does surface in curricula for universities of similar caliber to Trinity University, it is more often applied than it is examined from a theoretical perspective. However, the ACM curriculum guidelines for undergraduate degree programs in computer science recommend 18 content areas for study, one of which is Algorithms and Complexity. They suggest that students have familiarity with the standard complexity classes, familiarity with the-space trade-offs of algorithms, the ability to informally determine the time and space complexity of simple algorithms, and the ability to determine upper bounds on space and time complexity using big-O notation.

In addition to being endorsed by ACM, an examination of space complexity can be beneficial in heightening understanding of time complexity, the more often studied aspect of computational complexity. Space complexity is worth examining on its own merit as well; concerns about time constraints often take precedence now, but the world is changing quickly, and the world of technology even more so. New needs and restrictions arise all the time out of technologies that did not exist even a few years ago. The future is unpredictable, and making gains in the realm of space complexity may very well forge a path for new things to come. Aside from possible future practicalities, there is a mathematical beauty to the world of computational complexity and the set theory that holds it together; space complexity is an integral part of that woven fabric. For these reasons, this thesis suggests that space complexity is worth learning about and puts forth a pedagogical unit on the subject for use and adaptation.

The sources for space complexity content referenced in this thesis assume various levels of familiarity with certain methodologies, problems, and concepts; utilize disparate notation to express ideas; and have different foci that shape the inclusion and exclusion of certain concepts, as well as the flow of information. The components of review, notation, and focus have been intentionally selected and blended from these sources to create a reasonably accessible, consistent overview of complexity theory. The movement through the content is based on the model of the complexity hierarchy, a system for classifying groups of problems based on the resources they require, with a nod to time complexity but a focus on space complexity. Over the course of the lessons, the hierarchy is expanded and clarified as supplementary information aids in the process of making sense of it all.

This portion of the thesis was built using the Understanding by Design, or UbD, methodology. UbD is based on leading research in the field of pedagogy and is regularly utilized by Trinity University's Masters of Arts in Teaching program. It aims to help teachers create meaningful and comprehensive learning experiences for their students by looking at desired results, crafting ways for students to demonstrate the knowledge they have acquired in line with those results, and planning lessons that empower students to explore that knowledge. The student will work through the process of learning, demonstrating, and carrying knowledge forward in the usual manner, but planning in the reverse order ensures that the most important

part of the process, the knowledge that students should walk away with, is the basis for everything that happens in the classroom.

This method suggests first establishing learning goals that come in two varieties. The first are understandings: what are the essential ideas concerning the content that students should understand and feel comfortable working with? The understandings that the lessons in this thesis are based on, as well as all other aspects of the application of UbD to this content, can be found in the Unit Template. Learning goals also encapsulate essential questions prompted by the course. These are questions that may follow trains of thought inspired by class content but eventually point outward, to other disciplines and themes that transcend multidisciplinary schools of thought.

Answers to the question, “What do I want my students to take away from this class?” can be translated into assessment evidence. UbD offers two ways to measure student understanding and achievement towards the learning goals. The first are performance assessments, or authentic ways for students to synthesize class content. Authenticity is typically more open ended; allows for problem solving, deep thought, or creativity; and presents students with a problem that looks and feels like a situation that one might encounter in real life. The two performance assessments included in this thesis ask students to demonstrate understanding of course content and then to build on that understanding in meaningful ways. The second type of assessment evidence may vary in delivery, but is designed to regularly check for a basic level of understanding. In this thesis, these more regular forms of feedback are included as comprehension questions at the end of each lesson.

The next step in UBD is to design the learning plan. This is the structure and content that will fill class time and facilitate the learning portion of the class. The learning plan may include lectures, activities, seminar discussions, or other treatment of content. In this thesis, the learning plan is a series of four lessons. Lessons were broken up based on similarity of concepts to one another and the relationships among concepts. The first lesson is an introduction to terminology, a review of time complexity, and the idea of space complexity as well as an examination of a few space complexity classes. The second lesson takes a closer look at the idea of class completeness. The third focuses on sublinear space complexity. Finally, the last lesson takes care of any loose ends in the hierarchy that the other lessons built and briefly explores intra-class relationships.

Each lesson is formatted as a series of lecture notes. There were two lines of reasoning behind the choice to exclude activities, discussions, etc. in treating content. The first is that the unit materials created for exploration of space complexity are designed to be widely usable by undergraduate computer science instructors. As each instructor has their own unique teaching style, the content of this thesis is provided in lesson format in order to be easily adaptable. The instructor may progress through each lecture word for word, or the instructor may use the lessons as a guide for recommended flow of content, helpful visuals, etc. The second is that helpful educational tools such as activities and discussions would be better selected by the instructor, who will have important contextual information such as class culture, size,

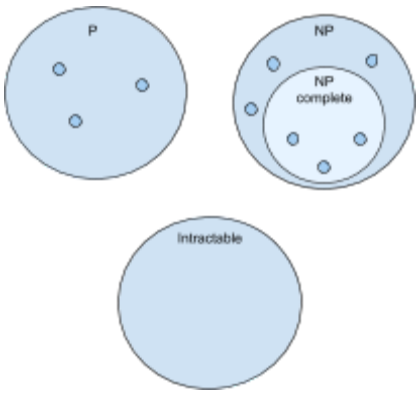
prerequisite knowledge base, etc. Sipser's *Introduction to the Theory of Computation* (2013) is the recommended text for this course; a significant portion of the lessons follow the development as presented by Sipser.

<table>
<tr><th colspan="2">Space Complexity Unit Template</th></tr>
<tr><td colspan="2">Established Goals
Explore the complexity hierarchy</td></tr>
<tr><td>Understandings
The complexity hierarchy can help us understand relationships among classes, as well as our (current) gaps in knowledge
The notion of complexity completeness is generalizable but only because we do not know the full nature of complexity relationships
Relationships among and within classes are formulated as hierarchy theorems</td><td>Essential Questions
How do space and time relate to one another computationally?
Are class distinctions contrived, natural, or are there some of each? Do they exist independently of our understanding?</td></tr>
<tr><td>Students will know… declarative knowledge - the facts, the what, the concepts
The complexity hierarchy
Naming conventions for classes in the hierarchy</td><td>Students will understand… procedural knowledge - the skills, the how to</td></tr>
<tr><th colspan="2">Assessment Evidence</th></tr>
<tr><td>Performance Tasks
Present the space hierarchy as you understand it
Consult on an imagined case study’s allotted CPU space for certain programs</td><td>Other Evidence
Exercises at the end of each lesson</td></tr>
<tr><th colspan="2">Learning Plan</th></tr>
<tr><td>Lesson 1: (3 days)
Review P, NP, NP-complete
Class complexity and problem complexity notation
Space complexity
Space constructible
PSPACE, NPSPACE
The “first relationship”
Savitch’s Theorem
Lesson 2: (1 day)
PSPACE-completeness</td><td>Lesson 3: (2 days)
Sublinear space complexity
The “second relationship”
L, NL, NL-complete, L-complete for P
Lesson 4: (2 days)
Space-constructible
Almost everywhere
Blum’s speedup theorem
Space Hierarchy Theorem and corollaries
Closing remarks</td></tr>
</table>

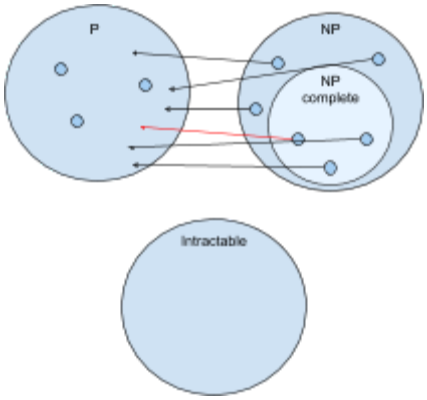
LESSON 1

Our next unit is on space complexity; we will get to what exactly that means in just a second. First, we will review what we have learned about time complexity.

Review: We have talked about P =? NP (Which is, more precisely, NP ⊂? P). Here is a quick recap.

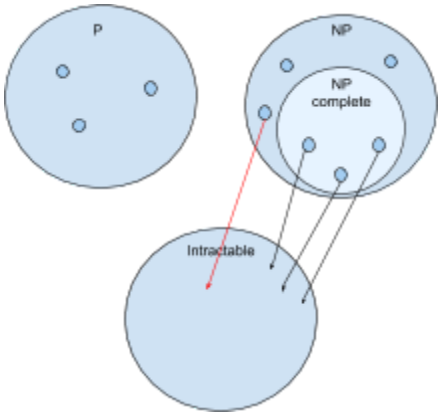


What we know currently. P ⊆ NP because all problems in P are nondeterministically solvable / checkable in polynomial time given a certificate (i.e., in NP). Certificates are helpful additional information (e.g., a proposed solution). We do not know if all problems that are checkable in polynomial time are also solvable in polynomial time. For problems in P, we simply disregard any certificate.



What we may discover (this would imply that P = NP): if even one problem in NP-complete is in P (the red arrow), then all problems in NP are in P (the black arrows).

… OR ...



What we may discover (this would imply that P ≠ NP, so P ⊂ NP): if even one problem in NP (but not necessarily in NP-complete) is intractable (red arrow), then all problems in NP-complete are intractable (black arrows). But the other problems in NP may be either in P or intractable.

Quick clarification: In this class, and generally, it is important to use set notation PROPERLY.

- $\subseteq$ means subset or equal to.
- $\subset$ means strict or **proper subset**: $A \subset B$ and $A \neq B$. A cannot be the same class as B if $A \subset B$.
- = means equal to. The two classes are the same class.

This may sound dumb, but it is important. Lots of people use $\subseteq$ loosely ("improperly": a pun!). The distinction between $\subseteq$ and $\subset$ will be of great importance to us in the ensuing discussion.

We have $P \subseteq NP$, where

- P refers to the class of all of the problems that can be solved in a polynomial number of steps as a function of input size *n*, or polynomial time.
- NP means nondeterministic polynomial. All of these problems can be *checked* in a polynomial number of steps, or polynomial time. But we are not necessarily sure if, as a class, they can be *solved* in polynomial time. Occasionally, we do find a polynomial solution for one.
- NP-complete problems are certain problems in NP that if they were solvable in polynomial time, this would mean that P = NP and thus $NP \not\subseteq EXPTIME$.

Note that EXPTIME is the class of all problems solvable in exponential time or better, while Intractable usually refers to all problems solvable in exponential time or worse. For our purposes, the idea of EXPTIME will be the more helpful term to examine and compare other classes to.

If you are not confident you understand the relationships here, please do some review or come talk to me or a friend for help. We are going to be using these concepts a lot.

Helpful Notation:

Throughout this unit, we will work towards developing a mathematical hierarchy to represent various interesting complexity classes. Our final result will look like this:

$L \subseteq NL \subset P \subseteq NP \subseteq PSPACE = NPSPACE \subseteq EXPTIME$; $NL \subset PSPACE$; $P \neq EXPTIME$; and $PSPACE \subset EXPSPACE$

This will mean absolutely nothing to most of you yet. But that is okay! We will keep revisiting this expression as we progress, and bit by bit it will make more and more sense. I promise. (We will show new discoveries in a **larger bold font**, and previously built knowledge in green.)

If time complexity is the number of (Turing machine) steps it takes to solve a problem, space complexity is the amount of space (Turing machine tape cells) it takes to solve a problem. You could compute this manually and in detail, which we will not do. While working programmers typically use bits/bytes as a common way of quantifying space, we will look at the amount of space a program takes up on a Turing machine: that is, tape cells. This will enable us to explore, in a general, machine-independent way, categories of space complexity and how they do or do not overlap.

When we talk about time complexity, we have broad categories like P, NP, and EXPTIME. We are going to broaden these ideas by extracting their fundamental pieces, then use those pieces to construct our space complexity classes. Much like with time, N will stand for nondeterministic (and the absence of the N will mean deterministic). We will have a symbol or set of symbols relating to the group of problems (P for polynomial, EXP for exponential, etc.). We will use the word SPACE to refer to space complexity classes; the absence of “SPACE” will refer to time complexity classes, as with “NP”.

Each problem will have one asymptotic notation to describe its time complexity and one to describe its space complexity. Each of these will belong to a certain complexity class based on

- Whether or not the computation of complexity is nondeterministic
- Whether or not the function describing complexity is polynomial, exponential, etc.
- Whether we are talking about time or space

So for example, PSPACE will signify a class of

- **Deterministic** (there is no “N”) ...
- **Polynomial** (solvable in polynomial **SPACE**) ...
- **Problems**.

Here is a visual aid that may help illustrate the relationships among these measures:

Building class notation: select one from each column

(determinism identifier)	(function type)	(space/time complexity identifier)
Either "N" or [blank]	P, L etc.	Either "SPACE" or [blank]

The second column represents examples of types of functions, but THIS IS NOT an inclusive list.

Note that "time" and "deterministic" are the defaults, so there is no signifier for those things! They are [blank]s. This discussion is for CLASSES and groups of classes.

The first diagram was about classes (groups of problems). Just as with time complexity, worst case space complexity of an INDIVIDUAL PROBLEM will be shown using asymptotic notation and can be identified in one of the classes shown before:

Building asymptotic notation: pick one from each column

(determinism identifier)	(complexity type identifier)	(function)
Either "N" or [blank]	Either "SPACE" or "TIME"	n^4, $log_2 n$, etc.

The following may help to clarify when one would use class vs. asymptotic notation:

Ex. 1 A certain problem may be computable by a "best" algorithm in n^4+n^2-1 steps. This is O(n^4).

- Asymptotic notation - This function runs in TIME(n^4).
- Class notation - This function is in the class P, since the algorithm is deterministic and solvable in polynomial time.

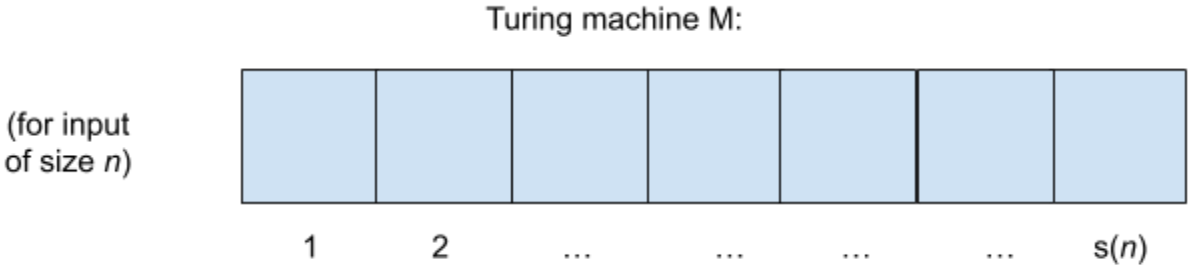
Ex. 2 A certain problem may be computable by a "best" nondeterministic algorithm on a Turing machine using n^8-225 tape cells. This is O(n^8).

- Asymptotic notation - This function runs in NSPACE(n^8)
- Class notation - This function is in the class NPSPACE, since the algorithm is nondeterministic and solvable in polynomial space.

Why do we group all of the polynomial, for example, functions together? Each function like n^3, n^5 or n^{456}, is its own class, but the collective group of polynomial functions is its own class too. For our purposes, it is most helpful to discuss the class that is the group of polynomial functions. (We will prove its existence later.) A key distinction for us is between those problems solvable in polynomial or better time/space versus those requiring exponential or worse time/space.

Discussing Space Complexity:

We still have not actually discussed what "space complexity" means, formally.



A Turing machine that runs in s(n) space needs at most s(n) tape cells to recognize a language. The language it recognizes is in SPACE(s(n)).

Formal definition: Let M be a deterministic Turing machine that halts on all inputs. The **space complexity** of M is the function s: N→N, where $s(n)$ is the maximum number of tape cells that M scans on any input of length n. If the space complexity of M is $s(n)$, we also say that M runs in SPACE($s(n)$).

Following is a brief example -- overview of how one might compute space complexity. Remember SAT? It is the Boolean satisfiability problem: given a certain Boolean expression, is the expression satisfiable (is there a certain way of assigning T/F (1/0) to the variables such that the expression evaluates as true)? It is an often cited NP-complete problem.

Here are the instructions for a Turing machine M that computes SAT:

"On input <Φ>, where Φ is a Boolean formula of n distinct variables:

- For each truth assignment to the variables x_1, …. x_n of Φ:
- Evaluate Φ on that truth assignment.
- If Φ ever evaluates to 1, *accept.* If not, *reject*.

This brute force method tries all possible truth assignments of all variables. However, we can reuse the same portion of Turing machine tape for each iteration of the loop. Assuming that there are n variables, each set of truth assignments only takes up $O(n)$ space -- i.e. only one bit per variable is needed to show true or false. So this machine runs in (exactly) SPACE(n).

It seems that space is more powerful than time, since space can be reused!

Like we did with SAT, we will be talking about space complexity in terms of how much space it would take a Turing machine to solve decision problems (because we are dealing with Turing machines, our measure of space will be the tape cell, not the byte traditionally thought of for physical machines). For the most part, we will be more focused on the nature of certain complexity classes than on how to find the space complexity of different kinds of problems.

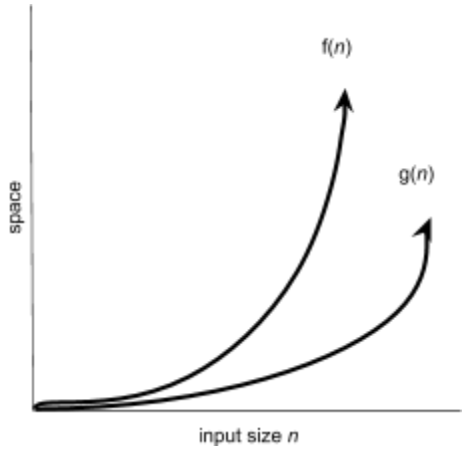
Space-Constructible Functions:

How do we distinguish classes? Polynomial (P) is a class, for example, containing all problems of $O(n^x)$ time complexity, where $x \in R$. Is a class just whatever we want it to be? We will discuss this more later, but for now we will focus on how we identify which classes are useful to us. Our classes are based on the idea of "well behaved" functions. Our technical term for this will be "space-constructible".

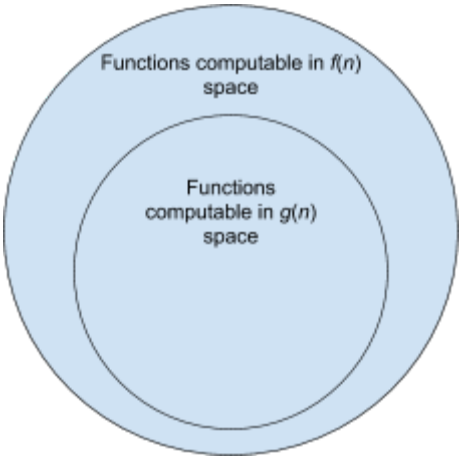
> A function $s(n)$ is **space-constructible** if there is some Turing machine M that is $s(n)$ space bounded, and for each n there is some input of length n on which M uses exactly $s(n)$ tape cells (think =, not ≤).

This will include functions like $log_2(n)$, n, n^k, 2^n, and $n!$. There is a time complexity analog for this concept (time-constructible), but as we are focusing on space complexity we will not talk about that in this class.

Why do we need this? The idea of space-constructible functions helps ensure that for any two space bounds we discuss, $f(n)$ and $g(n)$, if $f(n)$ is asymptotically larger than $g(n)$ and both $f(n)$ and $g(n)$ are space-constructible, $f(n)$ space allows us to solve a larger number of problems than we could in $g(n)$ space.



implies



Ex. n space allows a Turing machine to compute more than a $log_2(n)$ Turing machine
Ex. $n!$ space allows you to compute more than you can in 2^n space (all space-computable functions).

When you do not guarantee that $f(n)$ is space-constructible, it is possible that there exists a $g(n)$ that is only smaller than $f(n)$ by a very small and hard to compute amount such that $f(n)$ space gives you no more computing power than $g(n)$. We will choose from these space-constructible functions to build our complexity hierarchy for this class, but the hierarchy as it is known to exist (outside this class) contains many more classes and subclasses, for example, linear space.

The classes we will focus on are the ones given the most space in mainstream literature because they

- are distinct and large enough to be useful and worth studying
- may have interesting implications for P =? NP
- give you a general sense of the space hierarchy (which is the goal of this unit), rather than getting bogged down in one area

Now that we understand:

- what space complexity means, and
- how we decide which groups of functions form classes, and
- our class notation and asymptotic notation,

we will look at some space complexity classes. And using our previous over-all ranking, we will indicate exactly where we now are:

$L \subseteq NL \subset P \subseteq NP \subseteq$ **PSPACE** = NPSPACE $\subseteq$ EXPTIME; NL $\subset$ PSPACE; P $\neq$ EXPTIME; and PSPACE $\subset$ EXPSPACE

Discussing PSPACE:

PSPACE is the class of languages that are decidable in polynomial space on a deterministic Turing machine.

- PSPACE = $\cup_k$SPACE(n^k)
- PSPACE contains all of the problems that, for an input of size n, require $\leq n^k$ tape cells for a Turing machine to decide them, for all $k \in \mathbb{N}$.

It is analogous to, although distinct from, P in time complexity. Membership in the class P indicates a problem's running time, while membership in the class PSPACE indicates a problem's running space.

Ex. TQBF (True Quantified Boolean Formula) is in PSPACE

What is TQBF? Recall that a Boolean formula is an expression that contains Boolean variables, the constants 0 and 1, and the Boolean operations $\wedge$, $\vee$, and $\neg$ (not necessarily in conjunctive normal form as is standard in discussing SAT). We will be using a more general type of Boolean formula. Also recall the universal quantifier $\forall$ (where $\forall x \Phi(x)$ means that for all values of x, the statement $\Phi(x)$ is true) and the universal quantifier $\exists$ (where $\exists x \Phi(x)$ means that for at least one value of x, the statement $\Phi(x)$ is true). TQBF differs from first order predicate logic in that it requires all variables to be fully quantified and the only values that variables in TQBF may take are 0 and 1.

When a statement contains several quantifiers, universe (domain) and order matter!

Ex. $\forall x \exists y [y > x]$ vs $\forall y \exists x [y > x]$ in the universe of natural numbers

Quantifiers can fall anywhere inside of a boolean statement, and the scope of a quantifier is limited to what is inside of the parentheses immediately following the quantifier. When all quantifiers appear at the beginning of the statement and apply to everything in the statement, the statement is in **prenex normal form**. Boolean formulas with quantifiers are called **quantified boolean formulas**. When each variable of a formula appears within the scope of

some quantifier, the formula is said to be **fully quantified**. A fully quantified boolean formula is always either true or false.

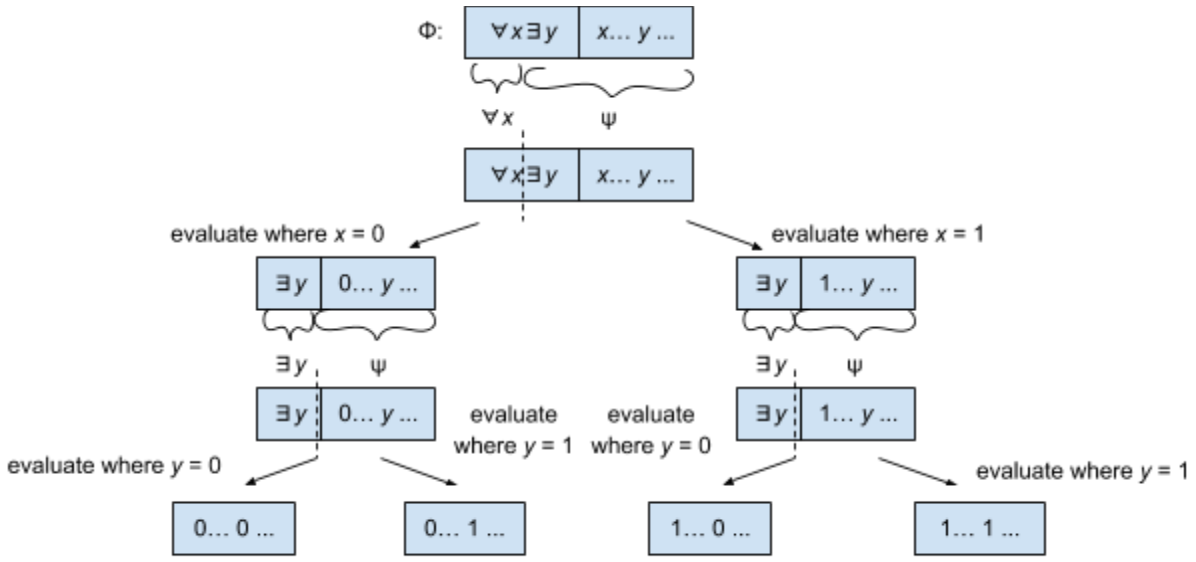
To determine if a certain fully quantified boolean formula in prenex normal form is true or false, we define the language.

TQBF: {<Φ>|Φ is a true fully quantified Boolean formula}

Now to prove that TQBF is in PSPACE. We will provide an algorithm that recognizes TQBF and prove that the algorithm is in PSPACE.

T: On input <Φ>,

- If Φ contains no quantifiers, then it only contains constants; so evaluate Φ and accept if it is true, otherwise reject
- If Φ equals $\exists x\psi$, recursively call T on ψ twice, once where $x = 1$ and once where $x = 0$. If either result is accept, then accept. Otherwise, reject.
- If Φ equals $\forall x\psi$, recursively call T on ψ twice, once where $x = 1$ and once where $x = 0$. If both results are accept, then accept. Otherwise, reject.



Stop when all of the quantifier portion has been cut off, meaning there are no variables left to instantiate.

The breadth of the recursion is not of concern in terms of space complexity, because for each evaluation of variable instantiation combinations the machine may reuse the same tape cells. The depth of recursion here is at most the number of variables. At each level we need only store

the value of one variable, so the total space used is O(m), where m is the number of variables in Φ. So the algorithm runs in linear space, which is within PSPACE.

∎

Now that we have treated PSPACE, we will approach the idea of NPSPACE.

L ⊆ NL ⊂ P ⊆ NP ⊆ PSPACE = **NPSPACE** ⊆ EXPTIME; NL ⊂ PSPACE; P ≠ EXPTIME; and PSPACE ⊂ EXPSPACE

Discussing NPSPACE:

Recall that we may specify the complexity of individual problems (whether space or time complexity) using asymptotic notation given a problem's deterministic nature and the function that best describes the order of magnitude of the desired complexity measure. The following definition is built off of the idea of NSPACE(n^k), which refers to the problems with nondeterministic algorithms to solve them using O(n^k) space.

NPSPACE is the class of languages that are decidable in polynomial space on a nondeterministic Turing machine.

- NPSPACE = $\cup_k$NSPACE(n^k)
- NPSPACE contains all of the languages which, given a certificate (e.g., proposed solution), are checkable in polynomial space by a Turing machine.

NPSPACE is analogous to NP in time complexity. We will not go over an example of an NPSPACE problem right now, and it will be clear why later on (there is also a hint in the hierarchy that keeps showing up).

We know that space can be reused (just write over the tape cells you have already used!) and time cannot.

Additional Relationships:

In the remainder of the paper, we will use $s(n)$ generically to notate the space usage upon execution on an input of size n; and we will use $t(n)$ to designate time usage. We know that if a program takes $t(n)$ time (steps) to run, it cannot use more than $t(n)$ tape cells since you can only move at most one tape cell per time step. As per our definition of a Turing machine, the highest

number of tape cells that you could use in $t(n)$ steps is $t(n)$. Thus, numerically, $s(n) \leq t(n)$. This, combined with our previous knowledge about P, and NP, leads to a few notable relationships:

- $P \subseteq NP$
- $P \subseteq PSPACE$
- $NP \subseteq NPSPACE$

As we go along, we'll uncover more relationships that will help us organize everything into our **complexity hierarchy.** These relationships form the basis for our end goal:

$L \subseteq NL \subset \mathbf{P \subseteq NP \subseteq}$ PSPACE $=$ NPSPACE $\subseteq$ EXPTIME; $NL \subset$ PSPACE; $P \neq$ EXPTIME; and PSPACE $\subset$ EXPSPACE

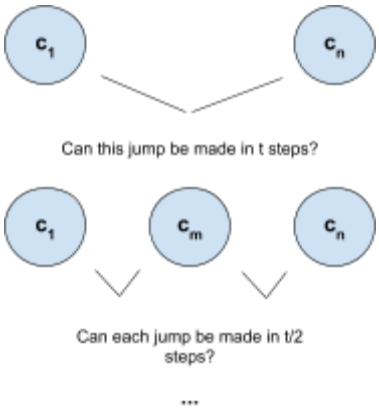
Recall that a Turing machine *configuration* is a complete snapshot of all parts of the machine -- the state, the location of the read/write head, and the contents of the tape cells:

configuration c = uqv where
(1) the tape is uv (u,v are strings over the tape alphabet);
(2) the state is q;
(3) the head is scanning the first (leftmost) symbol of v.

This definition of configuration can be extended to a more complex type of Turing consisting of multiple heads. With multiple heads, as we will see later, we will be able to investigate a phenomenon called sublinear space complexity.

The relationships we have uncovered so far hint that there is a discernible relationship between time and space. **We can bound the time complexity of a Turing machine in terms of its space complexity.** This is quantifiable in mathematical terms and will help us as we dive deeper into the hierarchy. I will refer to this as the "**first relationship**" due to its foundational importance. It is as follows: A program (Turing machine) that runs in $s(n)$ space can have at most $s(n)2^{O(s(n))}$ configurations for all $s(n)>n$.[1]

The idea here is that a Turing machine that halts cannot repeat a configuration. Note that this is not quite true for nondeterministic Turing machines. In that case, given a sequence of configurations *describing a halting computation*, any subsequence c_i ,..., c_j (where $c_i = c_j$) may simply be deleted since those steps in the computation were pointless and unnecessary! Everything (state, head position, input tape) is exactly the same at $step_j$ as it was at $step_i$. So, for this class, since we only care about halting Turing machines / decidable problems, we can assume that none of our Turing machines will repeat configurations.



[1] So where did the exponential base 2 come from (we are not necessarily in a binary environment)?
We're considering a Turing machine M; let v = |Q| and w = |Σ| (states and tape alphabet, respectively); both are constant with respect to M. Now, let the set of distinct, non-repeating configurations be C. Then
|C| = (# of states) • (# of tape cells) • (# of distributions of Σ symbols across the tape).
∴ $|C| = v \cdot s(n) \cdot w^{s(n)}$ by a standard combinatorial argument. In asymptotics we can disregard the constant v so that $|C| \approx s(n) \cdot w^{s(n)}$.
By simple logarithm / exponent manipulation, we can change the base from w to 2:
Consider the last term above: $w^{s(n)} = 2^{\log_2 w^{s(n)}}$.
$\log_2 w^{s(n)} = s(n) \cdot \log_2 w$; so that now, taking the inverse$_2$: $w^{s(n)} = 2^{s(n) \cdot \log 2(w)} = 2^{O(s(n))}$ since $\log_2 w$ is a constant. Hence, $|C| = s(n)2^{O(s(n))}$, as desired.

Discussing Savitch's Theorem:

Next, we will look at another relationship that seems to connect more clearly to the hierarchy we are aiming to build.

The **yieldability problem** will help us understand the proof for Savitch's theorem (below): given two configurations of a Turing machine, c_1 and c_2, together with a number t, can the Turing machine get from c_1 to c_2 within t steps?

To solve the yieldability problem, we will search for an intermediate configuration c_m, and recursively test whether c_1 can get to c_m in $\leq t / 2$ steps and whether c_m can get to c_n in $\leq t / 2$ steps. If, for some instance of the yieldability problem, a path can be found from c_1 to c_n using intermediate configurations this way in t or fewer steps, then that instance of the yieldability problem is true.

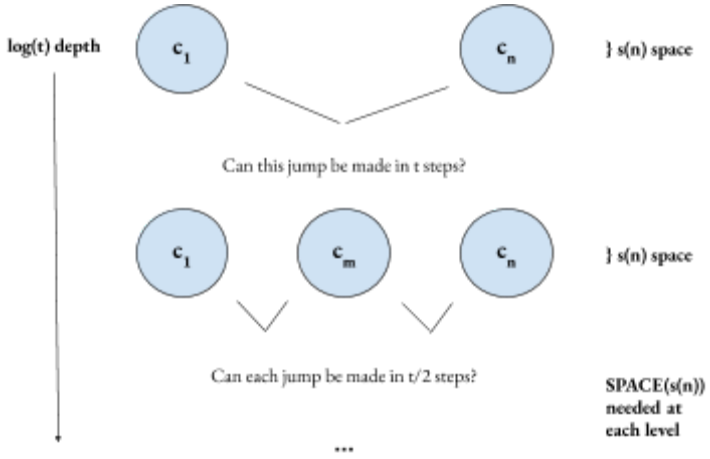
With this in mind, we will examine Savitch's Theorem.

Savitch's Theorem: For any function s: N -> R^+, where $s(n) \geq n$, NSPACE ($s(n)$) $\subseteq$ SPACE($s^2(n)$).

Proof idea: take an arbitrary Turing machine M that runs in NSPACE($s(n)$). To prove Savitch's theorem, we will simulate an equivalent, deterministic Turing machine D; this will help us to find a relationship between NSPACE and SPACE.

For every start configuration that leads to an accept configuration in M, we must ensure that the equivalent start configuration leads to an accept configuration in D. To check this, we will use an instance of the yieldability problem where

- c_1 is an arbitrary start configuration of M
- c_n is an arbitrary accept configuration of M
- t is the maximum number of steps that the deterministic Turing machine can use



To prove Savitch's Theorem holds, give a deterministic algorithm that yields c_n from c_1 in t steps (modeled after the yieldability problem).

We can reuse space for each of the two recursive tests. The algorithm needs space for storing the recursion stack. Each level of the recursion uses O($s(n)$) space to store a configuration. For this algorithm, by the first relationship, a branch that uses $s(n)$ space may run for $t = 2^{O(s(n))}$ time (steps). Then by taking the *log* of each side, the depth of the recursion is $log(t) = O(s(n))$, where t is the maximum time that the nondeterministic machine may use on any branch. Then our deterministic simulation uses $s(n) * O(s(n))$ space, or $O(s^2(n))$ space and is thus SPACE($s^2(n)$).

▮

It follows that for all n^k NSPACE (n^k) ⊆ SPACE(n^{2k}).

Since the square of a polynomial is still a polynomial, this gives us a polynomial-space conversion between nondeterministic Turing machines that use polynomial space and deterministic Turing machines that use polynomial space.

Corollary. Since you can convert between nondeterministic and deterministic Turing machines and stay within polynomial time, then PSPACE = NPSPACE.

▮

This is why we did not *explicitly* look at an example of a problem in NPSPACE! TQBF, our PSPACE example, also suffices as an example of a problem in NPSPACE. (The two spaces are the same.)

This gives us a bit more clarity in our complexity hierarchy. We started with

- P ⊆ NP
- P ⊆ PSPACE
- NP ⊆ NPSPACE
- PSPACE = NPSPACE

and can now rewrite them as P ⊆ NP ⊆ PSPACE = NPSPACE.

We are well on our way to developing our full hierarchy:

L ⊆ NL ⊂ P ⊆ NP ⊆ PSPACE = NPSPACE ⊆ EXPTIME; NL ⊂ PSPACE; P ≠ EXPTIME;

and PSPACE ⊂ EXPSPACE

Recall that for clarity in our hierarchy, classes and relationships that we have discussed together will appear in green, while the newest class or relationship that we discuss will appear in bolded, larger font.

Discussing EXPTIME:

This hierarchical relationship we have pieced together also supports the existence of a mathematical relationship between time and space… which is weird if you think about it. So far, it looks like space is bigger than time. It is odd to be quantifying time and space using the same metric (problems that can be solved within a given class' constraints). Certain classes may be "within" other classes, even if of the said classes, one is time and one is space.

Recall the "first relationship": a program that runs in $s(n)$ space can have at most $s(n)2^{O(s(n))}$ configurations for all $s(n) > n$. Therefore, since a Turing machine that halts must not repeat a configuration, a Turing machine that runs in SPACE($s(n)$) must run in TIME($s(n)2^{O(s(n))}$) for all $s(n) > n$; that is, at most $s(n)2^{O(s(n))}$ steps.

Because EXPTIME is mathematically defined as $\cup_k$ TIME(2^x) where $x = n^k$, and the given time constraint meets this definition (is within the class EXPTIME), we can say that a Turing machine that runs in SPACE($s(n)$) must run in EXPTIME $s(n) > n$. Furthermore, because $s(n)$ must be strictly linear or greater in terms of space needed to run, we may safely generalize our statement: a Turing machine that runs in PSPACE must run in EXPTIME.

If a Turing machine M runs in SPACE($s(n)$), then M must run in TIME($s(n)2^{O(s(n))}$) for all $s(n) > n$.	By the first relationship
If M runs in TIME($s(n)2^{O(s(n))}$) for all $s(n) > n$, then M is within the class EXPTIME.	By the definition of EXPTIME: $\cup_k$ TIME(2^x) where $x = n^k$
It follows that if a Turing machine M runs in SPACE($s(n)$), then M is within the class EXPTIME.	
If something is true for SPACE($s(n)$), then the same is true for PSPACE.	PSPACE is a subset of SPACE($s(n)$), for all functions $s(n) > n$
Therefore, if a Turing machine M runs in PSPACE, then M is within the class EXPTIME.	

Note that EXPTIME is not the same as Intractable. EXPTIME is a general, inclusive class that fits into our growing hierarchy; whereas Intractable problems are completely apart (complementary) to the tractable problems. Hence, EXPTIME is more in line with our (mathematicians') usual striving for increasing generality.

From this we can derive NPSPACE ⊆ EXPTIME (because PSPACE = NPSPACE, this is equivalent to saying PSPACE ⊆ EXPTIME). Note that our definition of EXPTIME is the class of all problems whose best solutions run in exponential time **or better.**

L ⊆ NL ⊂ P ⊆ NP ⊆ PSPACE = NPSPACE **⊆ EXPTIME;** NL ⊂ PSPACE; P ≠ EXPTIME; and PSPACE ⊂ EXPSPACE

Note also that P ≠ EXPTIME (P ⊂ EXPTIME). There is at least one problem that takes exponential time to solve but that cannot be solved in polynomial time.[2]

[2] For more information, see Chapter 9 (Intractability) of the Sipser text.

L ⊆ NL ⊂ P ⊆ NP ⊆ PSPACE **=** NPSPACE ⊆ EXPTIME; NL ⊂ PSPACE; **P ≠ EXPTIME**; and PSPACE ⊂ EXPSPACE

Now that we have a few concrete examples of classes, let us go back to a question we posed earlier. Who decides what a “class” is? Do they exist, and we just discover them? Or do they only exist as constructions in our mind that help us understand the world? In other words, are “classes” naturally occurring organizations of complexity? [Open to class for discussion. No “answer”; this is more of a philosophical musing].

Also, P =? NP is not the only open question we have about complexity! Anywhere we see “⊆” there is possibly room for clarification as we are often led to ask: Are the two classes equal, or not?

Hopefully by now you have a sense of why it is a complexity hierarchy we are building, not just a space hierarchy, and why we are still talking about time complexity in a space complexity unit. They are very intertwined!

Check for Understanding:

1. What is the difference between $\subseteq$, $\subset$, and = ?
2. Explain what “P” means as a term of computational complexity (What does it stand for? And what three characteristics can we derive based on notation alone?)
3. Explain what “SPACE(n^2)” means as a term of computational complexity (What does it stand for? And what three characteristics can we derive based on notation alone?)
4. Distinguish between the types of notation used in #2 and #3. When would you use one or the other?
5. Distinguish between the following (how are they different, in terms of their definitions?): computational complexity, time complexity, and space complexity.
6. What does it mean for a function to be space-constructible? And why is this a useful distinction to make?
7. A Turing machine that uses SPACE($s(n)$) must run in time _____.
8. A Turing machine that uses TIME($t(n)$) must run in space _____.
9. Explain in your own words how Savitch’s theorem implies that PSPACE = NPSPACE (you do not have to restate the proof idea here - just show me you understand the gist of it).
10. What does it mean for PSPACE to equal NPSPACE?
11. In the portion of the hierarchy we have built so far, P $\subseteq$ NP $\subseteq$ PSPACE = NPSPACE $\subseteq$ EXPTIME, what does $\subseteq$ represent in terms of open questions?

Answer Key

1. $\subseteq$ means subset of (and *may* be equal to), $\subset$ means *proper* subset of (cannot be equal to), and = means strictly equal to.
2. P refers to **a class of problems**. The problems in this class are **deterministically** solvable in **polynomial time** or better.
3. SPACE(n^2) refers to the computational complexity **of a specific problem**. This problem is solvable **deterministically** in **polynomial space**.
4. Use the notation exemplified in #2 when speaking about a group/class of problems. Use the notation exemplified in #3 when speaking about an individual problem.
5. Computational complexity is the more general term and refers to the resources it takes to solve a problem; whereas time complexity refers specifically to the number of time steps it takes to solve a problem, and space complexity refers specifically to the units of space (tape cells) it takes to solve a problem.
6. What does it mean for a function to be space-constructible? And why is this a useful distinction to make? If a function $s(n)$ is space-constructible, then there is some Turing machine M that is s(n) space bounded, and for each n there is some input of length n on which M uses exactly s(n) tape cells. This qualification helps us ensure that when building our complexity hierarchy, larger functions give us more power (allow us to solve more problems).
7. A Turing machine that uses SPACE($s(n)$) must have no more than $s(n)2^{O(s(n))}$ configurations. Since a halting Turing machine may not repeat a configuration, we may say that a Turing machine that uses SPACE($s(n)$) must run in TIME($s(n)2^{O(s(n))}$) or better.
8. A Turing machine that uses TIME($t(n)$) must run in SPACE($t(n)$) or better, since the number of tape cells you visit cannot exceed the number of steps you take. $s(n) \leq t(n)$.
9. Use the yieldability problem to deterministically simulate a nondeterministic machine. Show that this can be done in SPACE($s^2(n)$). This relationship then is NSPACE (n) $\subseteq$ SPACE(n^2). It follows then that NSPACE (n^k) $\subseteq$ SPACE(n^{2k}). The square of a polynomial is still a polynomial, so then SPACE(n^{2k}) is within NSPACE (n^k). The containment goes both ways, so PSPACE = NPSPACE.
10. It means that every problem deterministically solvable in polynomial space is also nondeterministically solvable in polynomial space, and that every problem that is nondeterministically solvable in polynomial space is deterministically solvable in polynomial space.

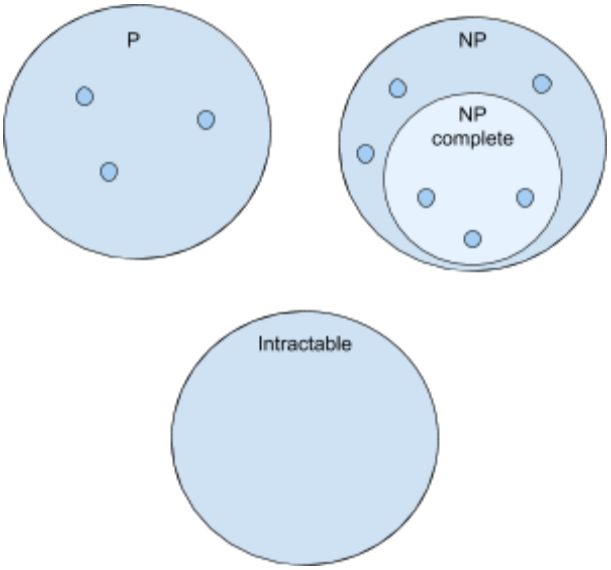
11. Here is the hierarchy we have built so far: P ⊆ NP ⊆ PSPACE = NPSPACE ⊆ EXPTIME, where ⊆ represents containment (i.e., the class on the left is within, and may be equal to, the class on the right of any given ⊆). In terms of open questions, ⊆ means that the two classes may be equal, or may not be; this is yet to be determined.

****

LESSON 2

In this lesson, we will explore the notion of “completeness” in complexity studies. First we will take a step back and review what we know about **NP-completeness**.

Review:

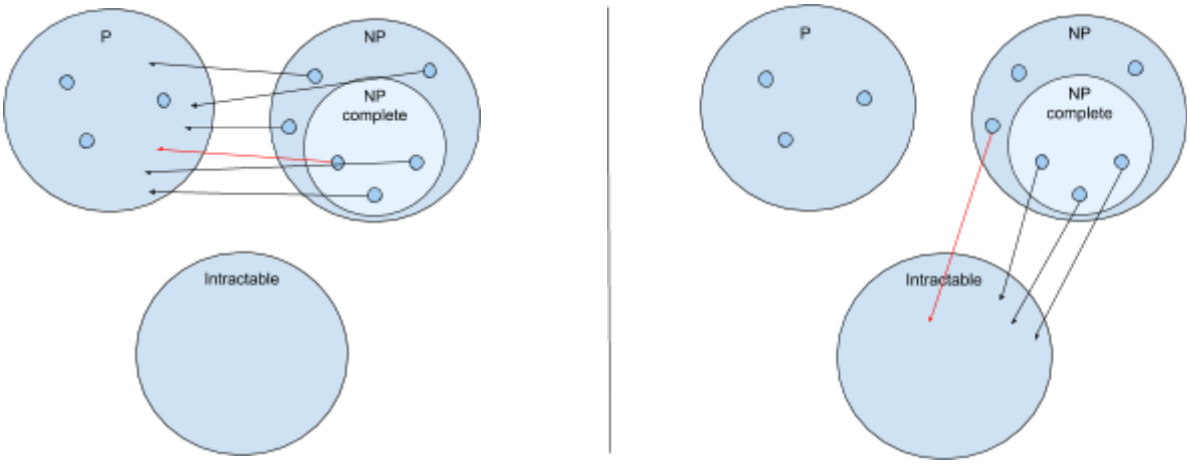


P is a class of problems that can be solved in polynomial time.

NP is a class of problems that, given a certificate (possible solution), can be checked in polynomial time.

NP-complete is a subclass of NP containing its most difficult problems. If any problem in NP-complete is solvable in polynomial time, then we know that all of NP is also solvable in polynomial time. (bottom left)

However, demonstrating that a language is NP-complete provides strong evidence that the language is not in P, since most think that all NP-complete problems are intractable.



Discussing Completeness in the Context of PSPACE-Completeness:

Now that we have reviewed NP-completeness, we can examine the concept of **PSPACE-completeness**. A language B is PSPACE-complete if

- B is in PSPACE
- Every A in PSPACE is polynomial time reducible to B.

Why does the definition use polynomial **time** reducible, and not polynomial **space**? Are we not talking about space? We should back up.

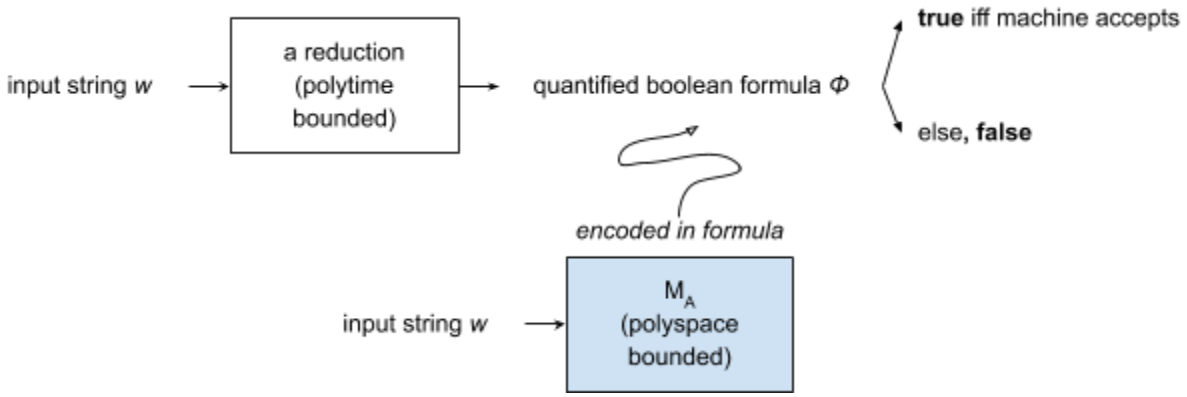
> "Complete problems are important because they are examples of the most difficult problems in a complexity class. A complete problem is most difficult because any other problem in the class is easily reduced to it (the notion of "easy" reduction will be expanded on later). So if we find an easy way to solve the complete problem, we can easily solve all other problems in the class. The reduction must be easy, relative to the complexity of typical problems in the class, for this reasoning to apply." - Sipser

We have mentioned before that space is more powerful computationally than time, since space can be reused. (Ex: A problem that takes exponential time may be solvable in linear space.) So an "easy" reduction needs to use a reasonable amount of time **and** space. Instead of stipulating both in the definition, we say "poly time" because that necessarily implies "poly space" (you cannot visit more tape cells than the number of steps you take!).

Example of a PSPACE-complete problem: TQBF
We learned previously that TQBF is in PSPACE, so all we need is to prove that:

Theorem. Every language A in PSPACE reduces to TQBF in polynomial time.
Proof idea: (The full proof can be found in Sipser's "Space Complexity" chapter.)

- Show TQBF is in PSPACE (we demonstrated this in Lesson 1)
- Show that every language in PSPACE reduces to TQBF in polynomial time
    - Start with a polynomial space Turing machine for an arbitrary A in PSPACE;
    - Give a poly time reduction that maps a string to a quantified boolean formula Φ that encodes a simulation of the machine on that input. The formula is true iff the machine accepts.



We have seen that “completeness” is not just a qualifier for the most difficult NP problems. There exists the concept of PSPACE-completeness too. Are there hardest/ “complete” problems for every complexity class? No, but there are for a lot of them![3]

Why is the notion of completeness useful?

- PSPACE-complete problems are likely not in NP (separates those two classes)
- NP-complete problems are likely not in P (separates those two classes)
- Complete classes give us a narrowed down set of problems (the hardest ones) from which to attempt to prove that certain $\subseteq$ relationships are actually just $\subset$. (For example, think back to the NP-complete problems. If even one of them is in P, then all of NP is also in P! (P = NP). By solving just one NP complete problem, we learn more about all of NP (and P!). Similar relationships exist for other complete classes.)
- Also supporting the notion of “hardest”, if even one problem in NP is intractable, then all of NPC is in EXPTIME!

[3] Generally, complexity classes that have a recursive enumeration have known complete problems, whereas classes that lack a recursive enumeration have none. For example, NP, co-NP, PLS, PPA all have known natural complete problems, while RP, ZPP, BPP and TFNP have no known complete problems (although such a problem may be discovered in the future). There are classes without complete problems. For example, Sipser showed that there is a language M such that BPP^M (BPP with oracle M) has no complete problems (Sipser, 1982).

Check for Understanding:

1. What does it mean for a problem to be PSPACE-complete?
2. Why is completeness a useful distinction to make?

Answers

1. If a problem is PSPACE complete, it is in PSPACE and every other problem in PSPACE is reducible to it in a polynomial or fewer number of time steps (and thus, tape cells).
2. Completeness, as a distinction, is useful for identifying the most difficult problems of a class. These problems can be useful in proving that two classes are equal or unequal (as in the case of P =? NP, and how if any NP-complete problem is in P, then P = NP).

To the instructor: At this point in the unit, introduce and assign Performance Assessment #1 to students. The knowledge base built in Lessons 1 and 2 should be sufficient background for this assessment. Students should be given the Performance Assessment #1 handout, the associated rubric to be submitted with their work, and the Project Feedback Sheet to be submitted with their work.

****

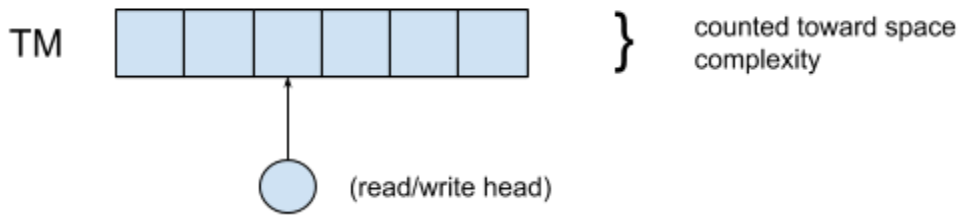
LESSON 3

Discussing Sublinear Space:

An important distinction we made between time and space is that space can be reused, while time cannot. In addition, the fact that we can only move one tape cell per time step ensures that the number of tape cells used will always be less than or equal to the number of time steps. An interesting phenomenon that arises is the idea of **sublinear space**. For this type of somewhat refined analysis, we ignore the input size and focus exclusively on the space needed for processing that input. To do this, we will modify our Turing machines:

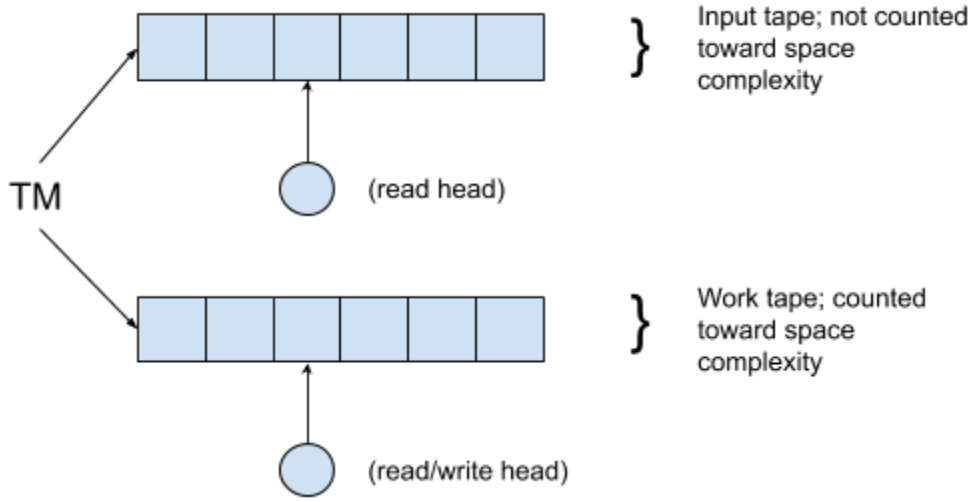
Original Turing machine

- One read/write tape



Our new, sublinear space Turing machine

- Two tapes: a read-only input tape, and a read/write work tape
- Only cells scanned on the work tape contribute to the space complexity of this type of Turing machine



This change allows our machine to manipulate data without storing all of it. It may seem like cheating; but imagine a computer and a CD-ROM - this happens more often than you would think! You may ask - well why do we not use this modification or a similar one in looking at time complexity? Can we not go sublinear there as well?

NO! This trick of only reading one input character at a time only works for space complexity. For an algorithm to be considered within a sublinear complexity class, it needs to be able to work while only storing one input character at a time. With time complexity, even if we read in one character at a time and placed it in the same tape cell, it would still require *n* steps to process all of the input.

Our first relationship (a Turing machine that uses space s(*n*) must run in TIME($s(n)2^{O(s(n))}$) for all s(*n*) > *n*) does not apply here, since s(*n*) < *n*. So what is our new relationship between time and sublinear space?

We will call this one the **"second relationship**": If M runs in s(*n*) space and *w* is an input of length *n* for *s*(*n*) < *n*, the number of configurations of M on *w* is $n2^{O(s(n))}$.

The second relationship simply substitutes the s(*n*) multiplier from the first relationship with *n*, since in the first relationship s(*n*) > *n*, and the roles are reversed with our sub-linear second relationship.

We will now define two sub-linear space complexity classes to add on to our hierarchy.

Discussing L and NL:

L is the class of languages that are decidable in logarithmic space on a deterministic Turing machine.

$$L = SPACE(log_2(n))$$

Notice that we use L, and not LSPACE. With P and PSPACE, P refers to polynomial time and PSPACE refers to its space analog. But with logarithmic bounds, there is no such thing as logarithmic time (reading in the input alone takes *n* steps, which pushes us above logarithmic

time bounds) so we do not need to make a distinction. The only option for a logarithmic bound is in regards to space, so L is a space complexity class.

NL is the class of languages that are decidable in logarithmic space on a nondeterministic Turing machine.

$$NL = NSPACE\ (log_2(n))$$

Why this class? Why not other sublinear classes like $\sqrt{n}$ or $log^2(n)$? These classes are also space-constructible; in working with these classes (and the functions that represent them), we know (as stated earlier) that classes based on larger (faster growing) functions are larger than classes based on smaller (slower growing) functions. Also, "logarithmic space is just large enough to solve a number of interesting computational problems, and it has attractive mathematical properties such as robustness even when the machine model and input encoding model change" (Sipser, p.349).

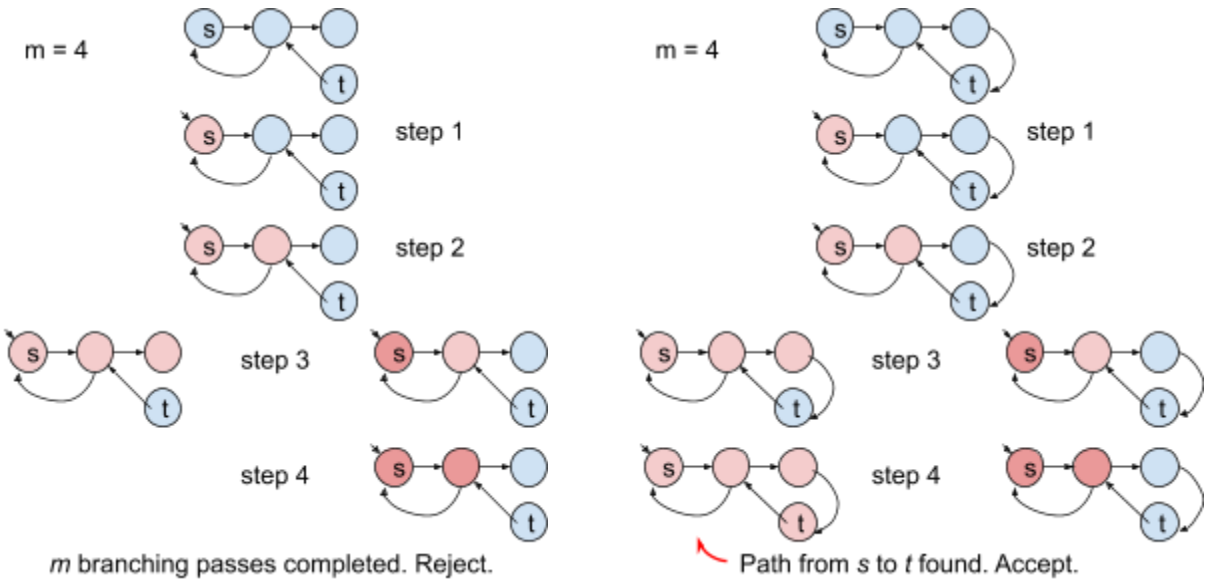
Example for $L = SPACE\ (log_2(n))$:

- The language $A = \{0^k1^k \mid k \geq 0\}$ is a member of L
- Proof: (construct an L-bounded deterministic Turing machine that recognizes A)
    - M: on an input string *w*
        - Scan across the tape and reject if a 0 is found to the right of a 1
        - Count the number of 0's in binary on the work tape
        - Count the number of 1's in binary on the work tape
        - The only space required is the space to store the two binary counts of the *k*-strings; this count is $log_2 k$ so that M runs in logarithmic space.

∎

Example for NL = NSPACE ($log_2(n)$):

- PATH = {<*G, s, t*> | G is a directed graph that has a directed path from *s* to *t*}
- Proof: (provide an L-bounded nondeterministic Turing machine that recognizes PATH
    - Start at node *s*
    - Nondeterministically guess the nodes of a path from *s* to *t*
        - Select the next node from those pointed at by the current node
    - (the machine records only the position of the current node at each step on the work tape, not the entire path)
    - Repeat until the algorithm reaches node *t* and accept, or repeat until it has run for *m* branching steps and reject (*m* is the number of nodes in the graph).



Key: Blue circles represent unvisited nodes, while red circles represent visited nodes. The darker shade of red represents nodes that have been visited twice.

Note that if this were done deterministically, it would require massive backtracking, because in the worst case all possible paths would need to be checked. There may be infinitely many paths in a general directed graph of *m* vertices if the graph contains any cycles; storing that unbounded count exceeds $\log_2(m)$.

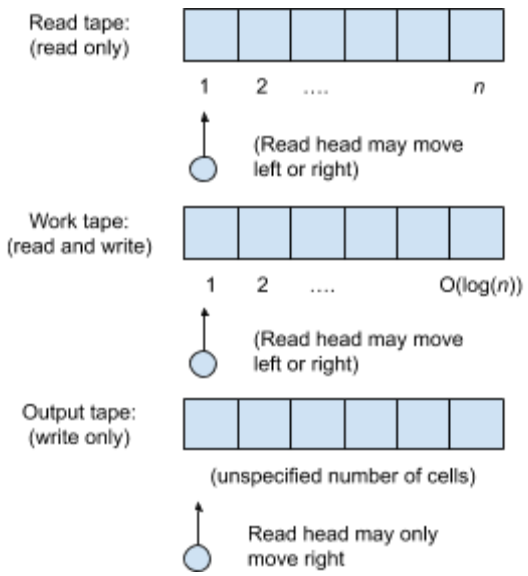
We know that PATH is in NL, but we are not sure if PATH is in L or not. So far, we do not know of any problem in NL that is **provably** outside of L. Analogous to the problem P =? NP We have L =? NL (written differently, that is L ⊂? NL).

This study shows us how our sublinear classes relate to one another, but not how they connect to the rest of the hierarchy quite yet:

L ⊆ NL ⊂ P ⊆ NP ⊆ PSPACE = NPSPACE ⊆ EXPTIME; NL ⊂ PSPACE; P ≠ EXPTIME; and PSPACE ⊂ EXPSPACE

In our approach to solve the problem of L =? NL, we encounter the NL-complete problems. These are the hardest problems in NL, meaning that all other problems in NL can be reduced to them. We have been using polynomial reductions in the past, but now we are looking at sublinear space classes and polynomial is not strict enough of a measure for reducibility.

Discussing Log Space Reducibility:



Let us define a log space transducer [a *transducer* is a machine that can generate a non-Boolean output] computing an input of size *n*, as a Turing machine with a read only input tape, a write only output tape, and a read/write work tape. The head on the output tape cannot move leftward, so it cannot read what it has written. The work tape may contain $O(log_2(n))$ symbols. A log space transducer M computes function $M: \Sigma^* \rightarrow \Sigma^*$, where $M(w)$ is the string remaining on the output tape after M halts when it is started with w on the input tape. We call M(w) a logspace computable function. Language A is log space reducible to language B, written $A \propto_{LOG} B$ if A is reducible to B by means of a log space computable function *f* (paraphrased from Sipser, p. 352).

We will use this in our definition of NL-complete:

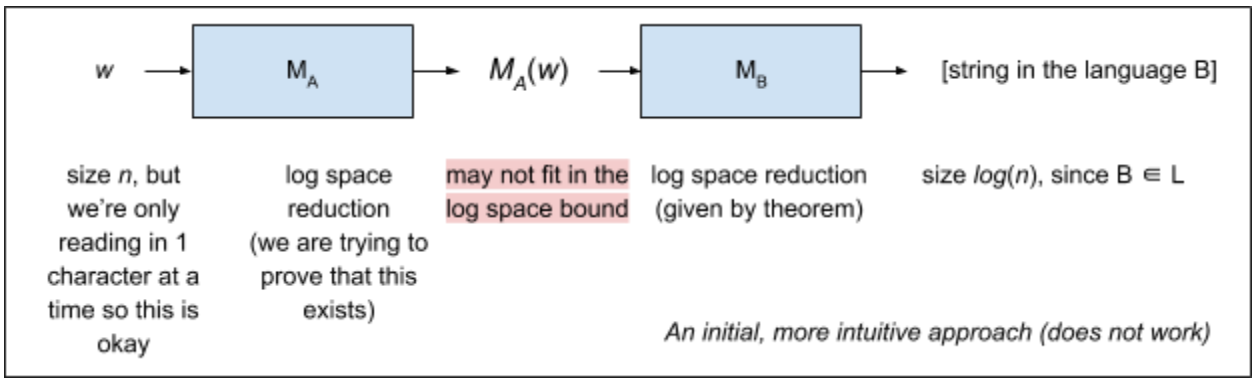
A language B is **NL-complete** if

- B ∈ NL and
- Every A in NL is log space reducible to B

In order to show NL-completeness, we cannot use the same method for proving polynomial time reducibility (it may take too much space). If we did that, the proof might look something like:

- First Turing machine A would map input *w* to $M_A(w)$ using the log space reduction M_A
- Then apply the log space algorithm for B

But the intermediary result $M_A(w)$ may be too large to fit within the logarithmic space bound, so to prove the theorem we need a different approach.

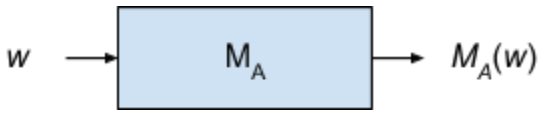


The highlighted note above indicates that this somewhat obvious approach will not work. So we will make the appropriate adjustment below:

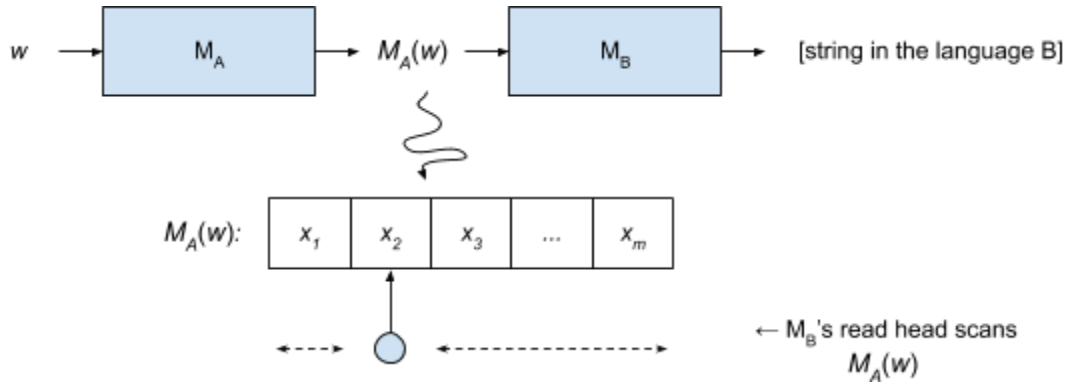
Theorem: If $A \propto_{LOG} B$, and B ∈ L, then A ∈ L

Proof:

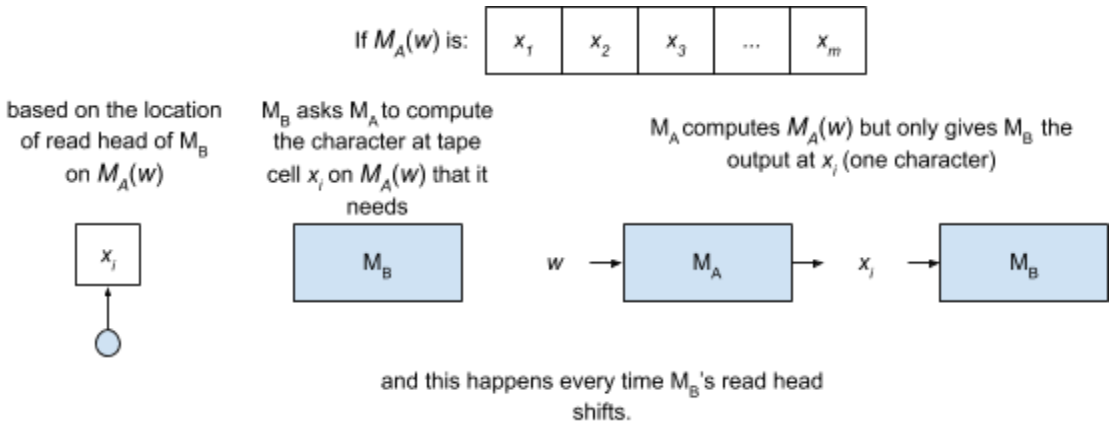
- We begin again with a log space algorithm for A maps its input *w* to $M_A(w)$, using the log space reduction M_A.



- A's Turing machine M_A computes individual symbols of $M_A(w)$ as requested by B's Turing machine M_B .



- In the simulation, M_A keeps track of where M_B 's input head would be on $M_A(w)$.
- Every time M_B moves, M_A restarts the computation of M_A on w from the beginning and ignores all the output except for the desired location of $M_A(w)$.



- Doing so may require occasional recomputation of parts of $M_A(w)$ and so is inefficient in its time complexity. The advantage of this method is that only a single symbol of $M_A(w)$ needs to be stored at any point, in effect trading time for space.

▮

Implications of Logspace Reducibility:

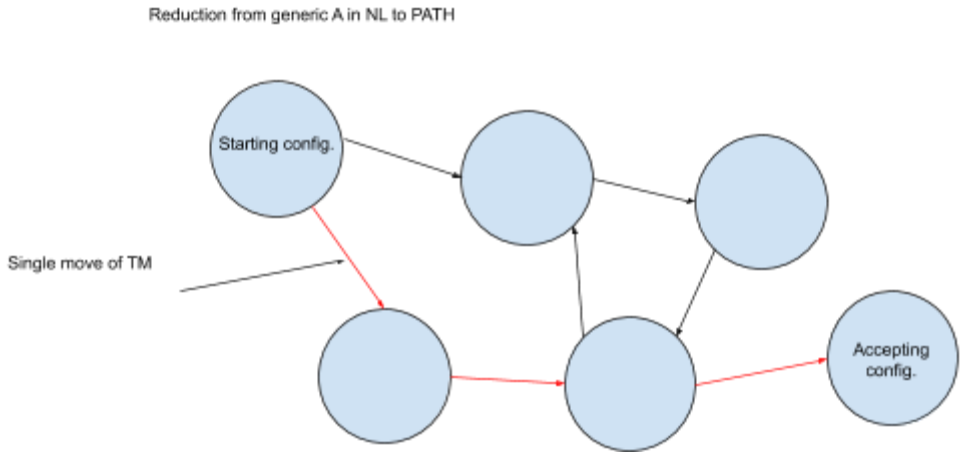
It follows that if any NL-complete language is in L, then L = NL. If a language is NL-complete, that means all other languages in NL are log space reducible to it (by the definition of NL-complete). Using the theorem above as a model: if A $\in$ NL, A $\propto_{LOG}$ B, and B $\in$ L, then A $\in$ L (meaning every language in NL is also in L, or L = NL).

Not only is PATH in NL, but moreover …

Theorem. PATH is NL-complete.

Proof Idea: We know that PATH is in NL, so we only need to show that every other problem in NL is log-space reducible to PATH. In other words, every language A in L is log space reducible to PATH. Recall that a *configuration* is defined as a complete snapshot of a Turing machine, where the contents of the tape, the location of any read or write head, and the state of the machine are represented.

Now we construct a graph that represents the computation of the nondeterministic logspace Turing machine for A. The reduction maps a string *w* to a graph whose nodes correspond to the configurations of the nondeterministic Turing machine on input *w*. One node points to a second node if the corresponding first configuration can yield the second configuration in a single step of the nondeterministic Turing machine. Hence, the machine accepts *w* whenever some path from the node corresponding to the start configuration leads to the node corresponding to the accepting configuration. The graph itself is an instance of PATH!



I

Corollary: NL ⊂ P

Indeed, “we know that any language in NL is log space reducible to PATH, since PATH is NL-complete. Recall that a Turing machine that uses SPACE(s(*n*)) runs in time $n2^{O(s(n))}$ (for sublinear s(*n*), according to the second relationship), so a reducer that runs in log space also runs in polynomial time [since $n2^{\log(x)} = nx$]. Therefore any language in NL is polynomial time reducible to PATH, which in turn is in P . We know that every language that is polynomial time reducible to a language in P is also in P (obviously!), so the proof is complete” (Sipser, p. 354).

I

This corollary shows us the link between the sublinear classes and the rest of our hierarchy. L ⊆ NL ⊂ P ⊆ NP ⊆ PSPACE = NPSPACE ⊆ EXPTIME; NL ⊂ PSPACE; P ≠ EXPTIME; and PSPACE ⊂ EXPSPACE

We have seen that there are a lot of open questions in regard to complexity. So why is P =? NP the only one we hear about?

- P =? NP is a question about the ease of solving problems. Problems that are in P are solvable in a reasonable amount of time, while intractable/EXPTIME problems cannot be solved in a reasonable amount of time. There are quite a few problems (ex. Travelling Salesperson) that people have stumbled upon in the real world that are provably NP-complete. Finding polynomial solutions for them would save lots of time, money, etc. If we discover that they are intractable, we can stop wasting our time looking for polynomial solutions. In such cases we would strive to develop approximation or probabilistic algorithms.
- Speeding up a program or helping it to use less space is always a good thing, but in the other cases of ambiguity (ex. L =?NL) solving an open question may not always lead to less space or time being used. Often in an attempt to decrease space needed, time resources needed increases, and vice versa.
- Also, some classes contain more widely relevant problems than others. People who do not know much about CS theory might have to deal with an NP-complete problem, but for lots of the other classes, we have to invent toy or highly theoretical examples to study those classes.

Can answers to these other open questions inform P =? NP? Potentially! We know that of all of the $\subseteq$'s in our hierarchy, at least one of them needs to be a strict containment ($\subset$) because P $\neq$ EXPTIME. So if we find out someday that NP = PSPACE and that NSPACE = EXPTIME, then P $\subset$ NP. While an interesting use of our hierarchy, it is probably the case that this method is not the most likely to provide a solution though. There are other approaches….

Discussing L-Completeness for P:

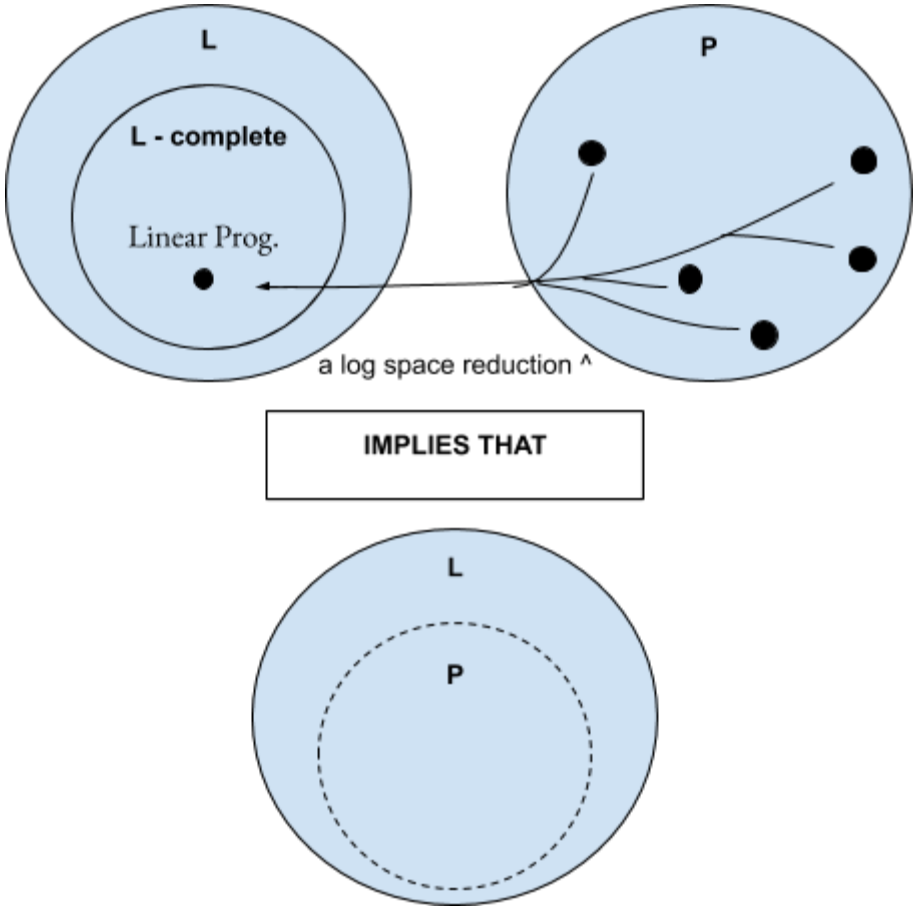
Not only do we have this concept of NL-completeness, but we also have log space completeness (for P).

A language L $\in$ P is **L-complete for P** if, for all L' $\in$ P, L' $\propto_{LOG}$ P.

For a language in P, the language is L-complete for P if all other languages in P are log space reducible to it. Why did we slice the cake like this? This seems like an odd distinction to make. Why? We will explore in a minute ….

Example of a language that is L-complete for P: Linear Programming Problem
The Linear Programming Problem is a maximization (minimization) problem that aims to maximize the value of a number (system) of linear functions (inequalities) where linear constraints are placed on their variables. Interestingly, though Linear Programming was once *thought* to be NP-Complete, it was ultimately *proved* to be in P (we will not prove that in this class) based on an algorithm found in 1979 (Linear programming, 2020). Linear Programming is L-complete (we will not prove that in this class either).

There are some interesting implications for log space reducibility, and especially log space reducibility for P. We established earlier that for a language in P, the language is L-complete for P if all other languages in P are log space reducible to it. Linear Programming is L-complete for P, so all other languages in P are log space reducible to Linear Programming. We know that Linear Programming is in P, but is there a log space algorithm for it (is it in L)? If we knew that Linear Programming $\in$ L, and all problems in P are log space reducible to Linear Programming, then we would know that P $\subseteq$ L (by the theorem above: If A $\propto_{LOG}$ B, and B $\in$ L, then A $\in$ L).



Then, from the hierarchy, we know that $L \subseteq P$. So, given the above, that would yield L = P! In our hierarchy, this would look like L = NL = P.

IF we can fine a logspace algorithm for Linear Programming, **THEN** $L = NL = P \subseteq NP \subseteq PSPACE = NPSPACE \subseteq EXPTIME$; and $P \neq EXPTIME$.

Keep in mind, all of this is ONLY IF we can find a log space algorithm for Linear Programming.

Log space transformations are more widely useful than they appear. Most (if not all) transformations useful in proving NP-completeness results are also log space transformations. All the transformations used for proving PSPACE-completeness are also log space transformations. Log space transformability also, as we saw before, can help us address the question of L =? NL.

Check for Understanding:

1. What does it mean for the space complexity of a problem to be sublinear? How is it possible for us to use fewer tape cells than there are characters of the input *n*?
2. Can time complexity be sublinear? Why or why not?
3. All other problems in NL can be reduced to the NL-complete problems. Why can we not use a polynomial reduction here, as we have before?
4. In your own words, how might Linear Programming be used to collapse/clarify our complexity hierarchy?

Answers:

1. That means that given an input of size *n*, the problem can be solved using $O(log_2(n))$ units of space. We can use fewer tape cells than there are characters of the input *n* by modifying the machine to include a read only tape that will store the input (not counted towards the space complexity), and reading the input onto the original read/write tape (counted towards the space complexity) one character at a time, writing over the old character with the new one each time so that only one total tape cell needs to be reserved for the input.
2. Time complexity cannot be sublinear. Reading each character of the input, whether it is done all at once in different cells, or one at a time in the same cell, will always take *n* steps, which is already linear.
3. Because in order to stay within logarithmic space bounds, the reduction needs to be logarithmic. If we use a polynomial reduction, the “reduction” of the problem in NL may no longer be logarithmic (within the class NL), and the reduction is no longer helpful in converting the problem to an NL-complete problem.
4. Because Linear Programming is L-complete for P, all other problems in P are log space reducible to Linear Programming. If we could prove that Linear Programming is in L, that would tell us that $P \subseteq L$. Since we already know that $L \subseteq P$, this would imply that $P = L$.

****

LESSON 4

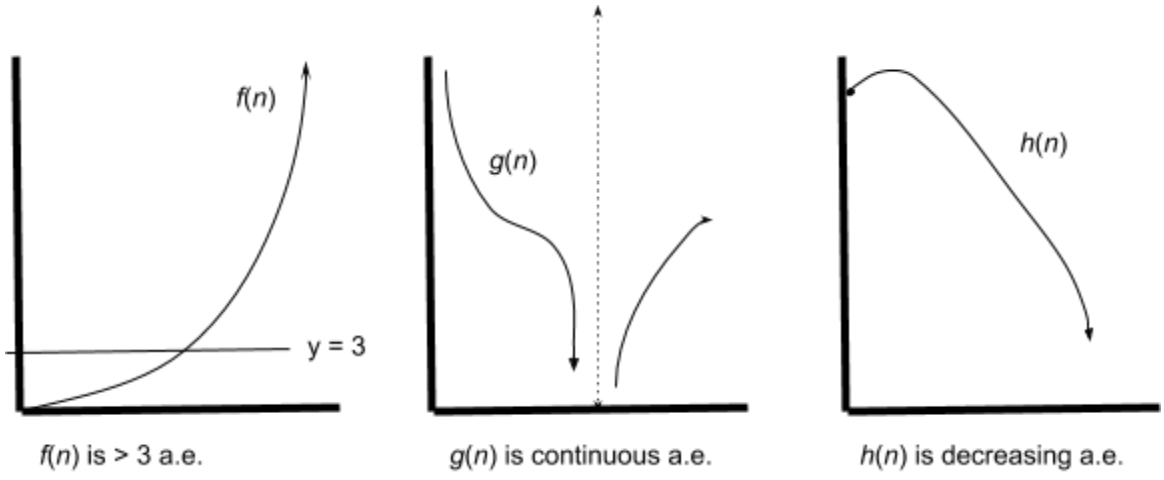
This last lesson in our computational complexity unit takes a closer look at relationships among and within classes. However, we will not introduce new classes, so this is a bit of a change of pace. We will look at two theorems relevant to our complexity hierarchy, although many more interesting theorems exist. We will focus on the Speedup and Space Hierarchy Theorems.

Restricting ourselves to the space-constructible functions is going to be very important here. Recall that a function $s(n)$ is **space-constructible** if there is some Turing machine M that is $s(n)$ space bounded, and there is some input of length n on which M actually uses $s(n)$ tape cells (meaning exactly equal to $s(n)$, not less than or equal to $s(n)$, as we often do in speaking about computational complexity).

Discussing *Almost Everywhere*:

We are also going to introduce a *new* nuanced notion.
A statement with parameter n is true **almost everywhere** (a.e.) if it is true for all but a finite number of values n. [Hoping not to be misleading, the following are drawn as continuous functions. But here, the domain is actually N (not the Reals); i.e., the functions have values only for natural numbers on the X-axis. With this understanding then, one sees that, e.g., $g(n)$ is decreasing for only a finite number of arguments.]

Examples:



$f(n)$ is > 3 a.e. $g(n)$ is continuous a.e. $h(n)$ is decreasing a.e.

When we talk about functions growing faster than other functions in the computer science context, we are really using the notion of a.e. without naming it. This goes back to the earliest analysis of algorithms course you have taken.

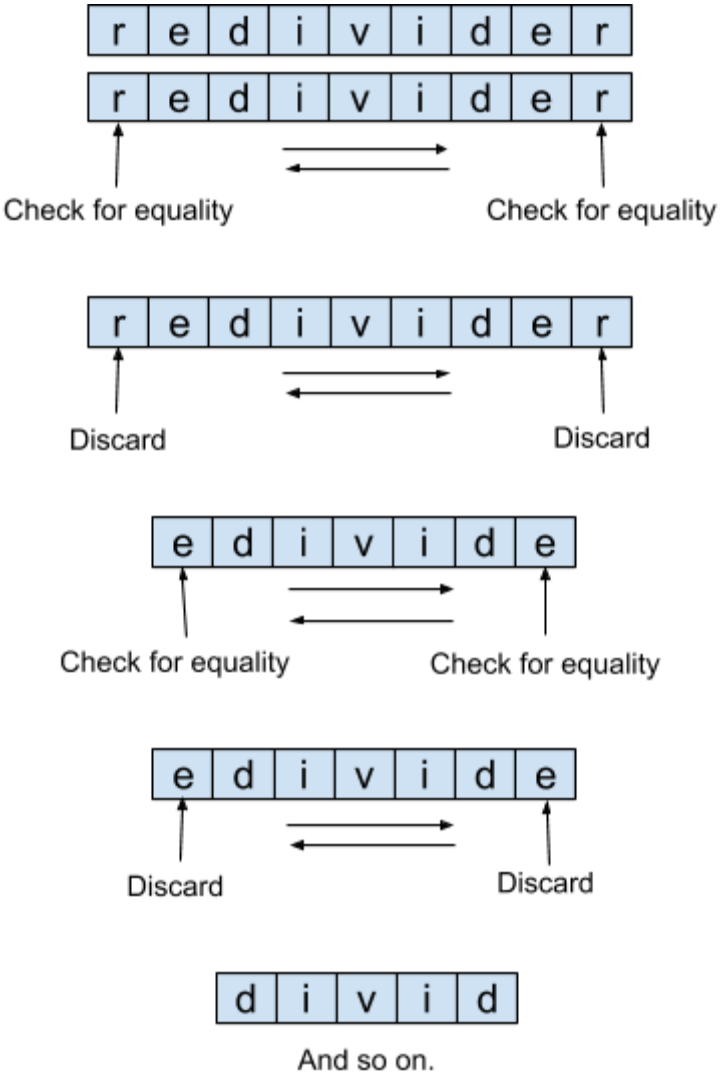
Discussing Blum's Speedup Theorem:

Now comes the *speedup theorem*. It postulates the surprising result that there exist functions with no best (most efficient) programs (Turing machines to recognize them). Recognizers for these programs can be sped up indefinitely! We will work with time here, but we could prove that this holds true for space as well.

Blum's Speedup Theorem: There are total computable functions *r* (i.e., defined for all elements of their domain) such that for any Turing machine M_i computing *r*, there exists a Turing machine M_j also computing *r* such that $T_j(n)) < T_i(n)$, for almost all *n*. Hence the computation of *r*(*n*) can be indefinitely *sped up*.

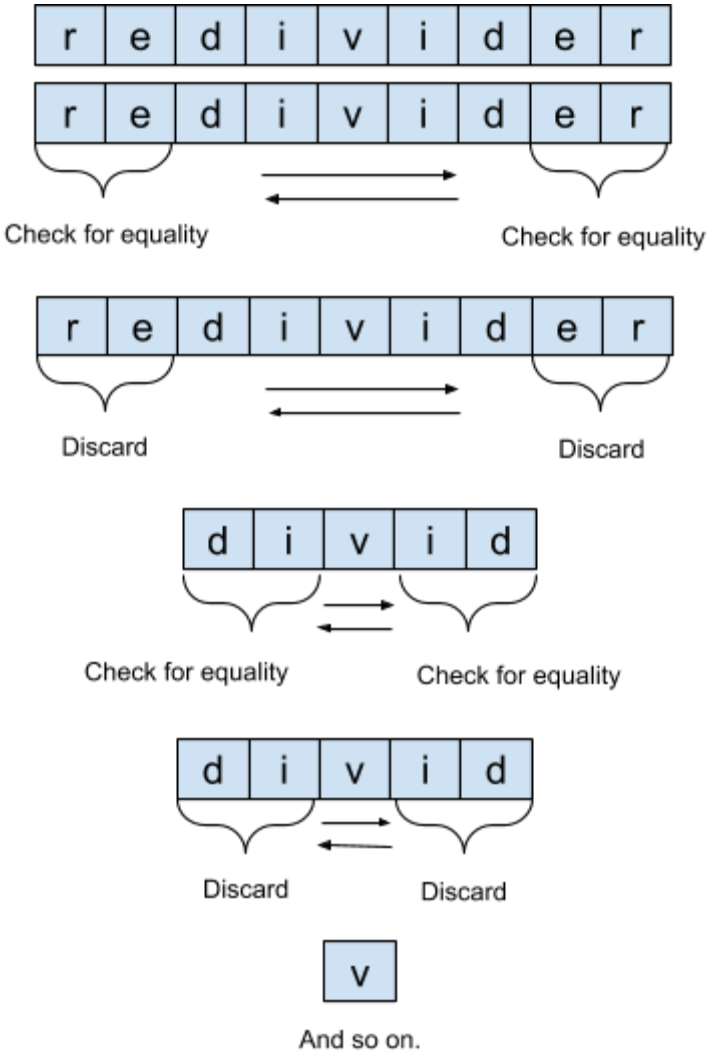
In other words, there are languages L such that if a Turing machine recognizes L, there is another Turing machine that also recognizes L and uses less time (space). You can look at Hopcroft and Ullman (1979) for the proof for this theorem, but here we will just go through an example to aid with understanding.

Example: Palindromes
Imagine a Turing machine whose language is exactly all palindromes (sequences of characters that are the same when read forwards or backwards). Without getting stuck in details, what might the algorithm for this Turing machine look like? [Open this time for discussion. The answer we are looking for involves comparing the characters on each end, removing them, and repeating.] That Turing machine might do something like this:

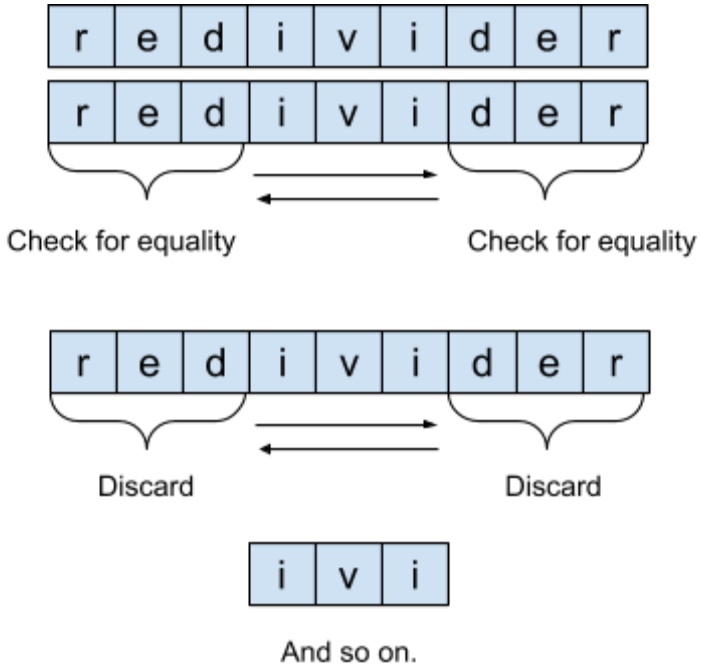


The time this algorithm takes is TIME(O(n^2)).

The Speedup Theorem tells us that there is a faster algorithm:



This algorithm takes about half the time of the original, because we are comparing two characters at a time instead of one. Here is an even faster algorithm:



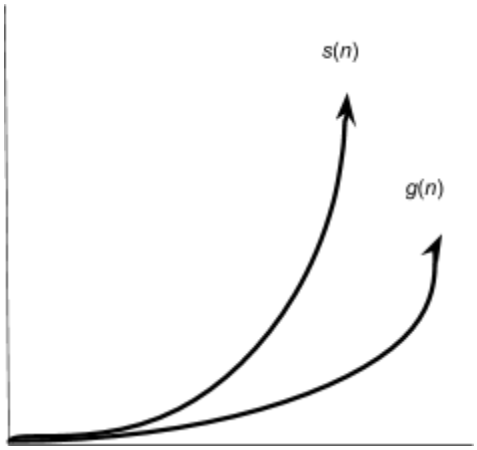
We can get faster and faster by checking more characters at once. Note that using this speedup method on this specific example, there is a limit as to how quickly we can ultimately speed up computation time. But the concept of unlimited speedup applies to the whole language, not just one example. Since our language of palindromes is infinite (and contains strings of arbitrary length), the concept of speedup, as it applies to the machine that accepts all of these strings, results in endless /unlimited improvement, exemplifying Blum's Theorem.

Looking back at these above examples that take less and less time, we may observe that they take less and less time by requiring more and more computational space! Using our Turing machine model, the first example that compares one character at a time only needs two tape cells on the work tape- one to store the first character, and one to store the last. The model that compares two characters at a time would require more space, to hold the four characters being compared and perhaps one to facilitate the reversal. This pattern continues. Our speedup comes at a cost. We are trading time for space here. Sometimes that is a worthwhile trade to make.

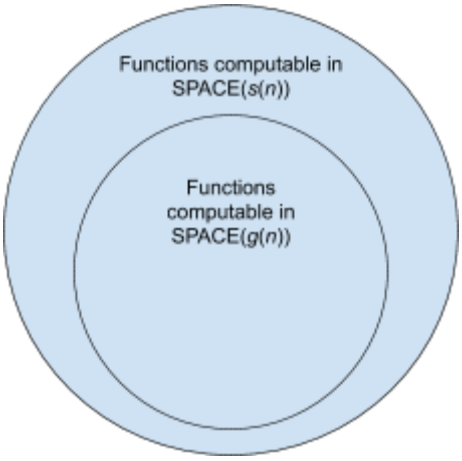
Other theorems, such as the Union Theorem and Gap Theorem, yield some interesting results about complexity theory but are outside the scope of this class.

Discussing the Space Hierarchy Theorem and its Implications:

But we are going to take a look at the **Space Hierarchy Theorem**. Recall that limiting ourselves to the space-constructible functions ($log_2(n)$, n, n^k, *etc.*) ensures that one space boundary $s(n)$ that is asymptotically larger allows us to solve more problems than would be possible within an asymptotically smaller space boundary $g(n)$.



implies



Before, I just told you that this was so, but now we are going to take a look at the proof outline and look at some other implications. We are talking about this Space Hierarchy Theorem because it yields some really cool results in regards to our hierarchy (as one would expect by its very name!).

Recall that if some $s(n)$ is space-constructible, some $O(s(n))$ space Turing machine exists that always halts with the binary representation of $s(n)$ on its tape when started on input 1^n. Recall that big-O is an asymptotic upper bound (could be equal to the function), while little-o is a strict asymptotic upper bound (s must be strictly less than $o(s(n))$).

Theorem (Space Hierarchy): For any space-constructible function $s: N \longrightarrow N$, a language A exists that is decidable in O(s(n)) space but not in o(s(n)) space.

Proof outline:

We must

- Demonstrate that a certain language A is decidable in O(s(n)) space.
- Demonstrate that A is not decidable in o(s(n)) space.
- Design an algorithm that meets both requirements.

However, we will not perform the constructions here .

Corollary 1: For any two real numbers $0 \leq \epsilon_1 < \epsilon_2$, SPACE($n^{\epsilon 1}$) ⊂ SPACE($n^{\epsilon 2}$)

This allows us to build a hierarchy even within PSPACE. We will not add it to our official hierarchy for this class, but this is a good example of how complex this idea of space complexity is.

Another corollary to the SPACE Hierarchy Theorem helps us to *separate* two classes that previously, we thought may have been equal to each other.

Corollary 2: NL ⊂ PSPACE

Proof:

Savitch's theorem shows that NL ⊆ SPACE($log^2(n)$), and the space hierarchy theorem shows that SPACE($log^2(n)$) ⊂ SPACE(n). It follows that NL is a strict subset ⊂ of linear space, which means it must also be a strict subset of polynomial space (PSPACE).

We will not be talking about linear space as a complexity class, but you are welcome to derive its relationships on your own!

So now ...

L ⊆ NL ⊂ P ⊆ NP ⊆ PSPACE = NPSPACE ⊆ EXPTIME; **NL ⊂ PSPACE**; P ≠ EXPTIME; and PSPACE ⊂ EXPSPACE

We have one more corollary that will improve upon our hierarchy. This one proves the existence of intractable problems (decidable but intractable, because of space constraints).

Corollary 3: PSPACE ⊂ EXPSPACE

Proof:

Corollary 1 tells us that $0 \leq \epsilon_1 < \epsilon_2$, SPACE($n^{\epsilon 1}$) ⊂ SPACE($n^{\epsilon 2}$).

So it follows that SPACE(n^k) ⊂ SPACE($n^{log(n)}$) since $0 \leq k < log_2(n)$ [a.e.*].

And SPACE($n^{log(n)}$) ⊂ SPACE(2^n).

Thus, SPACE(n^k) ⊂ SPACE(n^2). PSPACE is fully contained within and unequal to EXPSPACE.

∎

* as discussed earlier in this Lesson.

L ⊆ NL ⊂ P ⊆ NP ⊆ PSPACE = NPSPACE ⊆ EXPTIME; NL ⊂ PSPACE; P ≠ EXPTIME;

and PSPACE ⊂ EXPSPACE

This is the first time we have touched on the idea of intractable space. Intractability exists in terms of both time and space.

Check for understanding:

1. What is a.e.?
2. Explain in your own words what the Speedup theorem is.
3. What is the significance of the Space Hierarchy Theorem?

Answers:

1. a.e. means almost everywhere, and if a statement is true a.e. it is true for all but a finite number of values of *n*.
2. The Speedup theorem says that there are algorithms (such as palindrome recognition) that, if they solve problems using a certain amount of time or space resources, there exists an algorithm that solves the same problem but requires fewer time or space resources.
3. The Space Hierarchy Theorem proves that the idea of space-constructible function is valid and gives us the tools we need to show that NL $\subset$ PSPACE and PSPACE $\subset$ EXPSPACE.

To the instructor: At this point in the unit, introduce and assign Performance Assessment #2 to students. Alternatively, this assessment may be treated as an ungraded learning opportunity or extra credit assignment at your discretion. If it will be used to give grade based feedback, per the rubric, this assessment should be graded based on effort and what the student knows, rather than knowledge that the student lacks or technical details (see rubric for more information). The knowledge base built in Lessons 1-4 should be sufficient background for this assessment. Students should be given the Performance Assessment #2 handout, the associated rubric to be submitted with their work, and the Project Feedback Sheet to be submitted with their work.

****

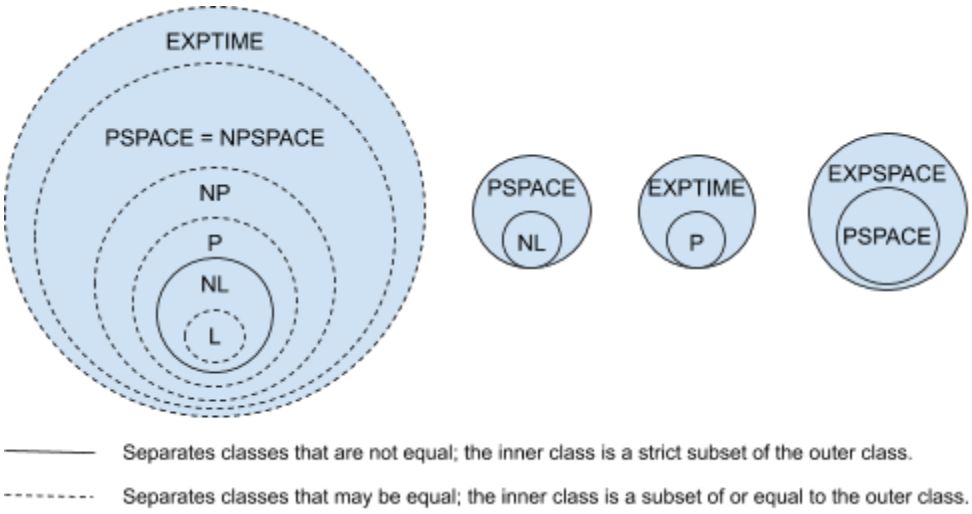
CLOSING REMARKS

To the instructor: Now that the space complexity unit/module is complete, these remarks are directed to the student. They represent a summing up of our previous lessons and guidance for further inquiry.

Now we have completed our task; we have developed the full complexity hierarchy as first presented. Moreover, our discussion has woven the notions of space- and time-complexity together into a somewhat comprehensive and intricate web.

$L \subseteq NL \subset P \subseteq NP \subseteq PSPACE = NPSPACE \subseteq EXPTIME$; $NL \subset PSPACE$; $P \neq EXPTIME$; and $PSPACE \subset EXPSPACE$

Here is an alternative, more visual way of expressing our hierarchy:



This format shows more clearly what open questions exist in terms of our hierarchy classes; any classes separated by a dashed line invite the curious theoretician to clarify the relationship. Are the classes equal, or is one a strict subset of another? The relative newness of this idea of space complexity and the wide array of problems yet to be solved suggests there are answers to be found for some or all of these questions. Questions regarding collapsing and separation of classes are by no means the only interesting questions to be answered in the field. Rather than focusing solely on classes and the problems whose solutions would have implications for

classes, one may choose to focus on a problem that is interesting on its own merit and search for an algorithm to solve it that is more efficient. With any *current* best time or space complexity for a problem comes a question: is there a faster algorithm? An example of this, as previously discussed, is the Linear Programming Problem. In 1979 it was proved to be in P. This improvement, although it had no implication on the separation or collapse of NP-Complete and P, now allows for any implementation of Linear programming to utilize fewer time resources. In a similar vein, one may desire to utilize the concept of space / time tradeoff in computation studied in the previous lessons to search for an algorithm that favors one at the expense of the other, based on whether lesser running time or running space is desired.

How might one approach searching for a more efficient solution to any of these open problems? This is where creativity comes into play. One may alter the current “best” algorithm, or consider an entirely new approach, such as the circuits approach suggested by Sipser’s Intractability chapter.

{Breaking News! Last month (October 2020) there was an announcement that an improved approximation for the Traveling Salesperson problem was discovered after decades of work (Klarreich 2020). This algorithm, while providing only miniscule improvement, uses a completely new methodology, thereby encouraging the continuing quest for improvements. Also, this discovery indicates the dynamic and exciting nature of these complexity studies!**}**

In general, I hope that this unit has impacted the way you view knowledge in general. Implicit in this discussion of space complexity is the idea that our collective knowledge is glaringly limited; we know at least a portion of what we do not know. And our intuition seems painfully incomplete, as we found that sometimes space is bigger than time, and sometimes it is not. Perhaps our pride is redeemed by the idea that these are problems that we have created for ourselves, that only exist because our minds are complex enough to create the situations in which these questions are worth asking. But this is a philosophical matter of perspective. Perhaps these mathematical and computational ideas exist outside of our desire or ability to consider them; there are many questions and answers waiting to be uncovered.

Performance Assessment #1

Explain the most important aspects of the complexity hierarchy in a way that makes sense to you. What was your biggest takeaway, or what was most meaningful to you?

Who: Your group (no more than three). This is a group assignment.

When: This project will be due on XX/XX/XXXX. You will have 1 week to work on this project.

Where: This project will be done outside of class.

Why: The purpose of this project is to showcase what you have learned about the space hierarchy and demonstrate how concepts discussed in class work together.

What: A friend of yours is taking an online course in space complexity and has learned all of the same things you have. They love the stuff but aren't quite understanding the bigger picture of how everything fits together. Using an essay, diagrams, powerpoint, a combination, and/or another format of your choice, showcase the major concepts from this class and how they relate to one another. Your end product should explore the most important aspects of the complexity hierarchy in a way that makes sense to you. What was your biggest takeaway, or what was most meaningful to you? Highlight that in your project!

Stuck? Here's an example: Why did we talk about these different classes? How are they the same and how are they different? What parallels do we see between relationships between different classes? What does completeness have to do with anything?

Your project may

- Include any other information you like
- Reference in-class materials and notes

Performance Assessment #1 Rubric (to be given to students with assignment handout, turned in with project, and returned with project)				
	Beginning	Developing	Accomplished	Exemplary
Thoughtful, meaningful connections made	**No attempt** is made to relate concepts from class to one another. (0-5 pts)	**An attempt is made** to relate concepts to one another, but **not much explanation** is given. (6-10 pts)	The product shows evidence of **thought-out synthesis**. Concepts are for the most part successfully related to one another. (11-15 pts)	The product shows ample evidence of **well developed synthesis**. Concepts are **clearly and successfully** related to one another. (16-20 pts)
Accuracy	**Many fundamental mistakes** demonstrate that the student holds deep misunderstandings of content. (0-5 pts)	Concepts from class and/or the connections between them are represented but with many mistakes. **Mistakes distract** the grader from the content of the project. (6-10 pts)	Concepts from class are **accurately represented** and connections are consistent with our current knowledge base, or only **mild errors** are present. (11-15 pts)	Concepts from class are **accurately represented** and connections are consistent with our current knowledge base. **No or a few errors** are present. (16-20 pts)
Formatting (overall)	Content is **not accessible** as a result of blurry text, coffee stains, etc. (0 pts)	Some formatting choices are **distracting**, but the content is still accessible. (1-2 pts)	Formatting choices are **suitable** to the project. The content is **easily accessible** to the grader. (3-4 pts)	Formatting choices make the project **easy to read and evaluate**. Formatting is **professional/ academic**. (5 pts)
Submission of feedback form	Not submitted or not taken seriously (0 pts)			Submitted and thoughtfully complete (5 pts)
				Total: _____ / 55

Performance Assessment #2 (groups of 2 or 3)

In light of trends towards machine learning and big data, companies are using more data. Terabytes and petabytes, as opposed to megabytes and gigabytes ("bytes" are just one way to make the abstract concept of space complexity concrete). Imagine that Tesla has lots of software and is looking to minimize storage allotted for various programs to run. They have used manual techniques (which we did not explicitly cover in this class) for estimating maximum storage needed by their biggest program in bytes, and have hired you as a consultant to explore the lower (or a lower) space bound for their program and (if that is possible) to further recommend whether that is something worth pursuing. They're asking you if it's worth pursuing an algorithm that takes less space, or if one even exists.

Here are three scenarios to think through:

1. Their program is currently considered to be run in SPACE(n^{965}), by their manual estimates of the maximum space bounds.
2. Their program is currently considered to be in SPACE(2^n), by their manual estimates of the maximum space bounds.
3. Their program is currently considered to be in class NL, by their manual estimates of the maximum space bounds.

You recommendation may

- Be in a format of your choosing
- Be informed by class notes and lecture materials
- Explore moving the problem to a different class, or moving it within its class
- Explore complexity coefficients
- Look at best case, worst, case, or average case
- Consider time/space tradeoffs

Your recommendation should include

- A discussion of the program in each scenario and whether or not you recommend looking for a more space efficient algorithm, including complications that arise and your justifications (Can it be done? Why or why not? Would the result be significant enough to justify the effort?).

Performance Assessment #2 Rubric
(to be given to students with assignment handout, turned in with project, and returned with project)

	Beginning	Developing	Accomplished	Exemplary
Understanding of core concepts	**No** recommendation is given **or little thought** is evidenced in the recommendation. (0-10 pts)	**A thoughtful attempt is made**, but there is evidence of **many fundamental misunderstandings** of core concepts. (11-20 pts)	There is evidence that the student has a **passable understanding** of core concepts. (21-30 pts)	There is evidence that the student **understands most or all** core concepts. (31-40 pts)
Providing answers in the context of the prompt	**No or little effort** is made to relate the discussion to the prompt. (0 pts)	**A thoughtful attempt is made**, but there is evidence of **lack of understanding** about how theoretical concepts may relate to the prompt. (1-2 pts)	There is evidence that **the student understands** how theoretical concepts may relate to the prompt. (3-4 pts)	The discussion of computational complexity is **successfully** given in terms of the prompt. (5 pts)
Formatting (overall)	Content is **not accessible** as a result of blurry text, coffee stains, etc. (0 pts)	Some formatting choices are **distracting**, but the content is still accessible. (1-2 pts)	Formatting choices are **suitable** to the project. The content is **easily accessible** to the grader. (3-4 pts)	Formatting choices make the project **easy to read and evaluate**. Formatting is **professional/ academic**. (5 pts)
Submission of feedback form	Not submitted or not taken seriously (0 pts)			Submitted and thoughtfully complete (5 pts)
				Total: _____ / 55

Project Feedback Sheet

(to be given to students with the project instructions and due when the project is due, but without student identifying information)

What about this project did you like? _____

_____

_____

_____

_____

Do you feel this project was a fair assessment of your knowledge and understanding in this class? Why or why not? _____

_____

_____

_____

_____

What about this project didn't you like? _____

_____

_____

_____

_____

What suggestions or comments do you have for future iterations of this project? _____

_____

_____

_____

_____

Conclusion of the Paper

The included pedagogical materials developed using the UbD methodology will be useful for teaching an undergraduate survey unit on computational space complexity. The unit materials may be adapted to fit the needs of an individual instructor or classroom.

Because of the design methodology used, the lessons provide an understanding of content to a depth and breadth consistent with questions asked of students at the end of each lesson, as well as the performance assessments that require deeper understanding for the purpose of synthesis. The performance assessments, in turn, will serve to indicate the instructor about where students are in terms of reaching the understanding content as laid out in the unit template, indicate to students how much or how little they understand as compared to instructor expectations and allow them to adjust accordingly without penalty, and invite students to think in ways that both enhance and transcend the discipline of computer science. The goal is learning and inquiry, and feedback in the form of grades is secondary to this. For this reason, lessons may be progressed through at a pace that feels comfortable to the instructor, with room to dwell on concepts that spark student interest or require additional explanation and room to pursue tangents that feel important or exciting (a few such possible directions are alluded to by footnotes or indicated as beyond the scope of the course in the lessons). Also in the spirit of supporting learning and inquiry, the rubrics for the learning assessments serve to provide students direction without stifling creativity. The grading metrics reward effort and understanding, rather than outlining too many specifics of things that must or must not be included. The project feedback sheet acknowledges that there is always room for improvement, even on the part of the academic content creator and instructor, and provides the student an outlet to make suggestions for improving the assessments for the next time they are used. In this iterative fashion, the assessments should evolve over time to better suit the needs of students. The glossary is provided for easy reference during lectures, homeworks, and completion of the assessments. As understanding is more valuable than memorization, making these more complicated terms quickly accessible encourages the student to focus on their application and meaning.

As stated before, the pedagogical components of this thesis offer the undergraduate computer science instructor material to offer students for the purpose of enhancing their understanding of space complexity and, indirectly, time complexity. This would enhance the well

roundedness of course offerings in computer science as suggested by the ACM Curricular Guidelines, especially for the peer institutions of Trinity University.

Glossary:

$\subseteq$: subset or equal to

$\subset$: strict or proper subset

= : equal to

Almost Everywhere : a statement with parameter *n* is true almost everywhere (a.e.) if it is true for all but a finite number of values *n*

Blum's Speedup Theorem: There are total computable functions *r* (i.e., defined for all elements of their domain) such that for any Turing machine M_i computing *r*, there exists a Turing machine M_j also computing *r* such that $T_j(n)) < T_i(n)$, for almost all *n*. Hence the computation of *r*(n) can be indefinitely *sped up*.

First Relationship : a program that runs in $s(n)$ space can have at most $s(n)2^{O(s(n))}$ configurations for all $s(n) > n$

Fully Quantified Boolean Formula : a boolean formula with quantifiers where each variable of the formula appears within the scope of some quantifier

L : the class of languages that are decidable in logarithmic space on a deterministic Turing machine; L = SPACE ($log_2(n)$)

L-complete for P : a language $L \in P$ is L-complete for P if, for all $L' \in P$, $L' \propto_{LOG} P$

NL : the class of languages that are decidable in logarithmic space on a nondeterministic Turing machine; NL = NSPACE ($log_2(n)$)

NL-complete : a language B is NL-complete if $B \in NL$ and every A in NL is log space reducible to B

NP : the class of all problems that can be checked in a polynomial number of steps (or polynomial time) as a function of the input size *n*.

NP-complete : the subclass of the hardest problems in NP; all problems in NP are polynomial reducible to problems in NP-Complete.

NPSPACE : the class of languages that are decidable in polynomial space on a nondeterministic Turing machine; NPSPACE = $\cup_k$NSPACE(n^k)

P : the class of all of the problems that can be solved in a polynomial number of steps (or polynomial time) as a function of input size *n*.

Prenex Normal Form : when all quantifiers appear at the beginning of the statement and apply to all variables in the statement, the statement is in prenex normal form

PSPACE : the class of languages that, for an input of size *n*, require $\leq n^k$ tape cells for a Turing machine to decide them, for all $k \in$ N; PSPACE = $\cup_k$SPACE(n^k)

PSPACE-completeness : a language B is PSPACE-complete if B is in PSPACE and every A in PSPACE is polynomial time reducible to B

Quantified Boolean Formula : a Boolean formula with quantifiers

Savitch's Theorem: for any function *s*: N → R^+, where $s(n) \geq n$, NSPACE ($s(n)$) ⊆ SPACE($s^2(n)$)

Second Relationship : if M runs in s(*n*) space and *w* is an input of length *n* for s(*n*) < *n*, the number of configurations of M on *w* is $n2^{O(s(n))}$**.**

Space Complexity : for a function s: N → N, where *s*(*n*) is the maximum number of tape cells that Turing machine M scans on any input of length *n*, the space complexity of M is *s*(*n*)

Space-constructible : a function *s*(*n*) is space-constructible if there is some Turing machine M that is *s*(*n*) space bounded, and for each *n* there is some input of length *n* on which M uses exactly *s*(*n*) tape cells

Space Hierarchy Theorem : for any space-constructible function *s: N → N*, a language A exists that is decidable in O(*s*(*n*)) space but not in o(*s*(*n*)) space

Yieldability Problem : given two configurations of a Turing machine, c_1 and c_2, together with a number *t*, can the Turing machine get from c_1 to c_2 within *t* steps?

Bibliography

Barrington, D. M., & Maciel, A. *Lecture 9: Hierarchy Theorems*. *Basic Course on Computational Complexity*. Buffalo; University at Buffalo.

Beckman, F. S. (1980). *Mathematical Foundations of Programming*, 1980, Addison-Wesley Publishing Co.

Catalogue: Computer Science Major. (2019). https://catalogue.vassar.edu/preview_program.php?catoid=33.

Computer Science Courses: Catalogue. (2020). https://www.colby.edu/catalogue/courses/CS/.

Computer Science Curricula 2013. (2013). https://www.acm.org/binaries/content/assets/education/cs2013_web_final.pdf.

Garey, M. R., & Johnson, D. S. (1979). *Computers and Intractability: A Guide to the Theory of NP Completeness*. Freeman and Co.

Hopcroft, J. E. & Ullman, J. D. (1979). *Introduction to Automata Theory, Languages, and Computation* (Ser. Addison-Wesley series in computer science). Addison Wesley Publishing Company, Inc.

(, October 8). Computer Scientists Break Traveling Salesperson Record. *Quanta Magazine*.

Linear programming. (2020, October 18). https://en.wikipedia.org/wiki/Linear_programming.

Massachusetts Institute of Technology. *Linear Programming Basics*. web.mit.edu. http://web.mit.edu/lpsolve/lpsolve-default/doc/LPBasics.htm.

McTighe, J., & Seif, E. (2011). A Summary of Underlying Theory and Research Base for Understanding by Design. https://www.jaymctighe.com/wp-content/uploads/2011/04/UbD-Research-Base.pdf.

Sipser, M. (2013). *Introduction to the Theory of Computation* (3rd edition). Cengage Learning (Boston).

Sipser, M. (1982, July 12). Lecture Notes on Relativization and the Existence of Complete Sets. https://link.springer.com/chapter/10.1007/BFb0012797.

Wiggins, G. P., & McTighe, J. (2005). *Understanding by Design* (2nd ed.). Association for Supervision and Curriculum Development.

Wikimedia Foundation. (2019, December 5). *Complete (complexity)*. Wikipedia. https://en.wikipedia.org/wiki/Complete_(complexity).