CS 120 Notes

Elvin Lo

Fall 2024

2024.12.20

Preface

These notes follow CS 120 (formally CS 1200) at Harvard College, taught by Professors Anurag Anshu and Salil Vadhan. A major accompanying text is Cormen and Leiserson, *Introduction to Algorithms*, Fourth Edition.

Contents

1 Sorting and General Computational Problems

1.1 The Sorting Problem

Definition 1.1. Sorting problem
Representing data items as key-value pairs (K, V), we can define the sorting problem as follows:

- **Input:** An array A of key-value pairs $((K_0, V_0), \ldots, (K_{n-1}, V_{n-1}))$, where each $K_i \in \mathbb{R}$.
- **Output:** An array A' of key-value pairs $\left((K'_0, V'_0), \ldots, (K'_{n-1}, V'_{n-1})\right)$ that is a valid sorting of A, i.e., a permutation of A that is sorted by key values (not necessarily unique).

1.2 Exhaustive-Search Sort

Algorithm 1.2. Exhaustive-search sort
Simply check every permutation $\pi : [n] \to [n]$, returning the first valid sorting.

Proof. To prove correctness, we must show that whenever there exists a valid sorting of A, then our algorithm will return one. To do so, suppose A has a valid sorting. Then there exists a permutation π such that $K_{\pi(0)} \leq K_{\pi(1)} \leq \cdots \leq K_{\pi(n-1)}$. Then our loop will halt and return some A' determined by some permutation π' (not necessarily π) in which the keys are ascending, as desired. □

1.3 Insertion Sort

Algorithm 1 Insertion sort

Input: An array $A = ((K_0, V_0), \ldots, (K_{n-1}, V_{n-1}))$, where each $K_i \in \mathbb{R}$
Output: A valid sorting of A

▷ *in-place sorting algorithm that modifies A until it is sorted* ◁
for $i = 0, \ldots, n-1$ **do**
  ▷ *insert $A[i]$ into the correct place among $(A[0], \ldots, A[i-1])$, s.t. $A[0:i]$ is sorted* ◁
  Find the first index j such that $A[i][0] > A[j][0]$;
  Insert $A[i]$ into $A[j]$ and shift $A[j \ldots i-1]$ to $A[j \ldots (i+1)]$
return A

Proof. To prove the correctness of insertion sort, observe that at the start of each iteration i, the continguous subarray $A[0 : i-1]$ is sorted. We may induct to show that this loop invariant is maintained at each iteration, and of course when the loop terminates then the entire array is sorted. □

1.4 Merge Sort

Algorithm 2 Merge Sort

Input: An array $A = ((K_0, V_0), \ldots, (K_{n-1}, V_{n-1}))$, where each $K_i \in \mathbb{R}$
Output: A valid sorting of A

```
procedure MergeSort(A)
    if n ≤ 1 then
        return A
    else
        i = ⌈n/2⌉
        A_1 = MergeSort(((K_0, V_0), ..., (K_{i-1}, V_{i-1})))
        A_2 = MergeSort(((K_i, V_i), ..., (K_{n-1}, V_{n-1})))
        return Merge(A_1, A_2)
```

Proof. The proof of correctness uses strong induction on the statement that MergeSort correctly sorts arrays of size n.

Now let us provide an analysis of the runtime of the algorithm. For $n \geq 3$, our definitions (given in Lecture 2) imply that

$$T_{\text{mergesort}}(n) = \max_{m \leq n} T^{=}_{\text{mergesort}}(m).$$

From the description of the Merge Sort algorithm, we find that

$$\begin{aligned} T^{=}_{\text{mergesort}}(m) &\leq T^{=}_{\text{mergesort}}(\lceil m/2 \rceil) + T^{=}_{\text{mergesort}}(\lfloor m/2 \rfloor) + T_{\text{merge}}(m) + \Theta(1) \\ &= T^{=}_{\text{mergesort}}(\lceil m/2 \rceil) + T^{=}_{\text{mergesort}}(\lfloor m/2 \rfloor) + \Theta(m) \end{aligned}$$

Here, $T_{\text{merge}}(m)$ is the runtime to merge two arrays of size at most m; we use without proof the fact that $T_{\text{merge}}(m) = \Theta(m)$. Since, $m \leq n$, we have $T^{=}_{\text{mergesort}}(\lceil m/2 \rceil) \leq T_{\text{mergesort}}(\lceil n/2 \rceil)$ and $T^{=}_{\text{mergesort}}(\lfloor m/2 \rfloor) \leq T_{\text{mergesort}}(\lfloor n/2 \rfloor)$. Thus, we obtain the following recurrence relation:

$$T_{\text{mergesort}}(n) \leq T_{\text{mergesort}}(\lceil n/2 \rceil) + T_{\text{mergesort}}(\lfloor n/2 \rfloor) + \Theta(n)$$

Solving such recurrences with the floors and ceilings can be generally complicated, but it is much simpler when n is a power of 2. In this case,

$$\begin{aligned} T_{\text{mergesort}}(n) &\leq 2 \cdot T_{\text{mergesort}}(n/2) + c \cdot n \\ &\leq 2 \cdot (2 \cdot T_{\text{mergesort}}(n/4) + c \cdot (n/2)) + c \cdot n \\ &= 4 \cdot T_{\text{mergesort}}(n/4) + 2c \cdot n \\ &\leq 4 \cdot (2 \cdot T_{\text{mergesort}}(n/8) + c \cdot (n/8)) + 2c \cdot n \\ &= 8 \cdot T_{\text{mergesort}}(n/8) + 3c \cdot n \\ &= \cdots \\ &\leq 2^{\ell} \cdot T_{\text{mergesort}}\left(n/2^{\ell}\right) + \ell c \cdot n \end{aligned}$$

for any natural number $\ell \leq \log n - 1$ (here and throughout CS 1200 all logs are base 2 unless otherwise specified). Note that the constant c in the above equations upper bounds $T_{\text{merge}}(n) = \Theta(n) \leq cn$. Taking $\ell = (\log n) - 1$, we get

$$T_{\text{mergesort}}(n) \leq \frac{n}{2} \cdot T_{\text{mergesort}}(2) + c(n \log n) = O(n \log n)$$

When n is not a power of 2, we can let n' be the smallest power of 2 such that $n' \geq n \geq \frac{n'}{2}$. Then

$$T_{\text{mergesort}}(n) \leq T_{\text{mergesort}}\left(n'\right) = O\left(n' \log n'\right) = O(n \log n)$$

The first inequality follows from the fact that we are taking the maximum running time over inputs of length at most n, so increasing n can only increase the maximum. The last equality follows from the fact that $n \leq n' \leq 2n$. The analysis can be done more carefully in a manner that one gets the following bound $T_{\text{mergesort}}(n) = \Theta(n \log n)$ (for example, by using the fact that there is another constant c' such that $T_{\text{merge}}(n) \geq c'n$). □

2 Computational Problems and Measuring Efficiency

2.1 Computational Problems

To study and compare different computational problems, we would like to formally define them. This will let us classify problems according to which ones have efficient algorithms (or no algorithms at all), relate different problems via reductions, and so on.

Definition 2.1. Computational problem
A computational problem Π is a triple $(\mathcal{I}, \mathcal{O}, f)$ where:

- $\mathcal{I}$ is a (typically infinite) set of possible inputs (a.k.a. instances) x, and $\mathcal{O}$ is a (sometimes infinite) set of possible outputs y.
- For every input $x \in \mathcal{I}$, a set $f(x) \subseteq \mathcal{O}$ of valid outputs (a.k.a. valid answers).

Informally, an algorithm is a well-defined procedure A for transforming any input x into an output $A(x)$.

Definition 2.2. Solving a computational problem
Algorithm A solves computational problem $\Pi = (\mathcal{I}, \mathcal{O}, f)$ if the following holds:

- $\forall x \in \mathcal{I}$ s.t. $f(x) \neq \emptyset, A(x) \in f(x)$
- $\forall x \in \mathcal{I}$ with $f(x) = \emptyset, A(x) = \perp$

2.2 Measuring Efficiency

Definition 2.3. Running time
Informally, for an algorithm A and input size function `size`, the (worst-case) running time of A is the function $T : \mathbb{R}^{\geq 0} \to \mathbb{R}^{+}$ given by:

$$T(n) = \max_{x \in \mathcal{I} : \texttt{size}(x) \leq n} \{\#\text{ basic operations when } A \text{ runs on } x\}$$

A variant of this is

$$T^{=}(n) = \max_{x \in \mathcal{I} : \texttt{size}(x) = n} \{\#\text{ basic operations when } A \text{ runs on } x\}$$

Definition 2.4. Asymptotic notations for complexity
Let $f, g : \mathbb{N} \to \mathbb{R}^+$. We say:

- $f = O(g)$ if $\exists c > 0, f(n) \leq c \cdot g(n)$ eventually (i.e., for sufficiently large n)
- $f = o(g)$ if $\forall c > 0, f(n) \leq c \cdot g(n)$ eventually, or equivalently if . . .
- $f = \Omega(g)$ if $\exists c > 0, f(n) \geq c \cdot g(n)$ eventually
- $f = \omega(g)$ if $\forall c > 0, f(n) \geq c \cdot g(n)$ eventually
- $f = \Theta(g)$ if $f = O(g)$ and $f = \Omega(g)$

For example, note that the running time $T(n)$ of exhaustive search sort satisfies both $T(n) = O(n! \cdot n)$ and $T(n) = \Omega(n! \cdot n)$, implying that $T(n) = \Theta(n! \cdot n)$. For an analysis on the runtime of merge sort, see the previous section.

3 Reductions

Definition 3.1. Reductions
Let $\Pi = (\mathcal{I}, \mathcal{O}, f)$ and $\Gamma = (\mathcal{J}, \mathcal{P}, g)$ be two computational problems. A reduction from Π to Γ is an algorithm that solves Π using as a subroutine an oracle that solves Γ. An oracle solving Γ is a function that, given any query $y \in \mathcal{J}$ returns an element of $g(y)$, or $\perp$ if no such element exists.

We denote:

- If there exists a reduction from Π to Γ, then we write $\Pi \leq \Gamma$.
- If there exists a reduction from Π to Γ which, on inputs (to Π) of size n, takes $O(T(n))$ time (counting each oracle call as one time step) and calls the oracle only once on an input (to Γ) of size at most $h(n)$, we write $\Pi \leq_{T,h} \Gamma$.
- If there is a reduction from Π to Γ that makes at most $q(n)$ oracle calls of size at most $h(n)$, we write $\Pi \leq_{T,q\times h} \Gamma$.

Lemma 3.2. Reduction lemma
Let Π and Γ be computational problems such that $\Pi \leq \Gamma$. Then:

1. If there exists an algorithm solving Γ, then there exists an algorithm solving Π.
2. If there does not exist an algorithm solving Π, then no algorithm solves Γ.
3. If there exists an algorithm solving Γ with runtime $R(n)$, and $\Pi \leq_{T,q\times h} \Gamma$, then there exists an algorithm solving Π with runtime
$$T(n) + O(q(n) \cdot R(h(n))).$$

The notation $\Pi \leq_{T,q\times h} \Gamma$ implies that solving Π requires runtime $T(n)$ (while considering the oracle queries to have constant time) with $q(n)$ queries to the oracle, where $h(n)$ is the input size to the oracle.

4 Static Data Structures

Definition 4.1. Static data structure problem
A static data structure problem is a quadruple $\Pi = (\mathcal{I}, \mathcal{O}, \mathcal{Q}, f)$ where:

- $\mathcal{I}$ is a (typically infinite) set of possible inputs x, and $\mathcal{O}$ is a (sometimes infinite) set of possible outputs y.
- $\mathcal{Q}$ is the set of possible queries.
- For every $x \in \mathcal{I}$ and $q \in \mathcal{Q}$, $f(x, q) \subseteq \mathcal{O}$ are the valid outputs on x, q.

A solution to a (static) data structure problem $\Pi = (\mathcal{I}, \mathcal{O}, \mathcal{Q}, f)$ is a pair of algorithms (Preprocess, Eval) such that

- for all $x \in \mathcal{I}$ and $q \in \mathcal{Q}$ such that $f(x, q) \neq \emptyset$, we have Eval(Preprocess(x), q) $\in f(x, q)$, and
- there is a special output $\perp \notin \mathcal{O}$ such that for all $x \in \mathcal{I}$ and $q \in \mathcal{Q}$ such that $f(x, q) = \emptyset$, we have Eval(Preprocess(x), q) $= \perp$.

Data structure problems are referred to as abstract data types and solutions are referred to as implementations.

Definition 4.2. Static Predecessors
Static predecessors are an important static data structure problem.

- **Input:** An array of key-value pairs $x = ((K_0, V_0), \ldots, (K_{n-1}, V_{n-1}))$, with each $K_i \in \mathbb{R}$
- **Queries:**
  - `search`(K) for $K \in \mathbb{R}$: output some (K_i, V_i) such that $K_i = K$.
  - `next-smaller` (K) for $K \in \mathbb{R}$: output some (K_i, V_i) such that $K_i = \max\{K_j : K_j < K\}$.

By simply sorting the array (i.e., taking a sorting method to be our preprocessing algorithm), we can then answer both the types of queries in $O(\log n)$ via binary search. As a remark, if we omitted the `next-smaller` queries, this data structure problem would just describe a static dictionary.

5 Dynamic Data Structures and Binary Search Trees

5.1 Dynamic Data Structures

Definition 5.1. Dynamic data structure problem
A dynamic data structure problem is a quintuple $\Pi = (\mathcal{I}, \mathcal{O}, \mathcal{U}, \mathcal{Q}, f)$ where:

- $\mathcal{I}$ is a (sometimes infinite) set of possible initial inputs x, and $\mathcal{O}$ is a (sometimes infinite) set of possible outputs y.
- $\mathcal{U}$ is a set of updates,
- $\mathcal{Q}$ is a set of queries, and
- for every $x \in \mathcal{I}, u_0, u_1, \ldots, u_{m-1} \in \mathcal{U}$, and $q \in \mathcal{Q}$, $f(x, u_0, \ldots, u_{m-1}, q) \subseteq \mathcal{O}$ is a set of valid answers.

Often we take $\mathcal{I} = \{\epsilon\}$, where ϵ is the empty input (e.g. a length 0 array), since the inputs can usually be constructed through a sequence of updates. A solution to Π is a triple of algorithms (Preprocess, EvalQ, EvalU) such that for all $x \in \mathcal{I}, u_0, u_1, \ldots, u_{m-1} \in \mathcal{U}$ and $q \in \mathcal{Q}$, if $f(x, u_0, u_1, \ldots, u_{m-1}, q) \neq \emptyset$, then $y_m \in f(x, u_0, u_1, \ldots, u_{m-1}, q)$, where $y_0, \ldots, y_m$ are defined inductively as follows:

- $y_0 = \text{Preprocess}(x)$, and
- $y_i = \text{EvalU}(y_{i-1}, u_i)$ for $i = 1, \ldots, m$.

If $f(x, u_0, u_1, \ldots, u_{m-1}, q) = \emptyset$, then we require that $y_m = \perp$.

Definition 5.2. Dynamic Predecessors
Dynamic predecessors are an important dynamic data structure problem.

- **Updates**:
  - `insert`(K, V) for $K \in \mathbb{R}$: add one copy of (K, V) to the multiset S of key-value pairs being stored (where S is initially empty).
  - `delete`(K) for $K \in \mathbb{R}$: delete one key-value pair of the form (K, V) from the multiset S (if there are any remaining).
- **Queries**:
  - `search`(K) for $K \in \mathbb{R}$: output $(K', V') \in S$ such that $K' = K$.
  - `next-smaller`(K) for $K \in \mathbb{R}$: output $(K', V') \in S$ such that $K' = \max\{K'' : \exists V''\,(K'', V'') \in S, K'' < K\}$.

Though a maintaining a dynamic sorted array is one possible implementation of a dynamic predecessor, a good approach achieving $O(\log n)$ time for each of these operations (and more) is a binary search tree.

5.2 Binary Search Trees

Definition 5.3. Binary search tree (BST)
BSTs are a recursive data structure, where the empty BST has no vertices. Every nonempty BST has finitely many vertices, including a root vertex r, and every vertex v has:

- a key K_v and value V_v,
- a left subtree, which is either the empty BST or is rooted at a vertex denoted v_L, which we call the left-child of v.
- a right subtree, which is either the empty BST or is rooted at a vertex denoted v_R, which we call the right-child of v.

We require that the sets of vertices contained in the subtrees rooted at v_L and v_R are disjoint from each other and don't contain v. Furthemore, we require that every non-root vertex has exactly one parent, and the root vertex has no parents. We denote the multiset S stored by a BST T as the multiset of key-value pairs occurring at vertices of T.

Crucially, we also require that the keys satisfy the BST Property: If v has a left-child v_L, then the keys of v_L and all its descendants are no larger than K_v, and similarly, if v has a right-child, then the keys of v_R and all of its descendants are no smaller than K_v.

Definition 5.4. Height of a tree
The height h of a BST is the length of the longest path from the root vertex to any leaf node. The height of an empty tree is defined to be -1.

The reason it makes sense to define the height of an empty tree to be -1 is so that the height of a tree is always one plus the maximum of the heights of its left-subtree and and right-subtree.

Theorem 5.5. Querying BSTs
Given a binary search tree of height (equivalently, depth) h, all of the dynamic predecessors/successors operations (queries and updates), as well as min and max queries, can be performed in time $O(h)$. That is, the queries are correctly answered according to the multiset stored by the tree, and the updates correctly modify the multiset stored by the tree while maintaining all the requirements of a BST (in particular, the BST Property).

See Lecture 5 for details on implementing these operations and analyzing their complexities.

5.3 Balanced Binary Search Trees

We would like to ensure that our binary search trees can remain shallow or balanced (e.g., of height $O(\log n)$ where n is the number of items currently in the dataset) when we update them. One approach for retaining balance is called AVL trees.

Definition 5.6. AVL Trees
An AVL Tree is a binary search tree in which:

- Every vertex has an additional attribute containing the height of the tree rooted at that vertex. ("data structure augmentation")
- Every pair of siblings in the tree have heights differing by at most 1 (where this includes the case where one sibling is the empty tree and has height -1). ("height-balanced")

Lemma 5.7.
Every AVL Tree with n vertices has height (depth) at most $2\log_2 n$.

Proof. Let $n(h)$ be defined as the min number of vertices in an AVL tree of height $\geq h$. Then, by the second AVL property, for $h \geq 2, n(h) \geq n(h-1) + n(h-2) + 1$. Since $n(h-1) \geq n(h-2)$ by definition, we can simplify this equation to

$$n(h) \geq n(h-1) + n(h-2) + 1 \geq 2n(h-2).$$

Now we are at a good place for unrolling the expression for even h (a similar argument holds for odd h):

$$\begin{aligned} n(h) &\geq 2n(h-2) \\ &\geq 4n(h-4) \\ &\geq 8n(h-6) \\ &\vdots \\ &\geq 2^{h/2}n(0) \\ &= 2^{h/2}. \end{aligned}$$

This shows that $h \leq 2\log n(h)$, which proves the lemma. □

In order to maintain the AVL property during insertion or deletion operations, we need define an additional operation rotation, which lets us move vertices around while preserving the BST property.

6 The RAM Model

6.1 Computational Models

Definition 6.1. Computational model
A computational model is any precise way of describing computations. It is a broad term, which encompasses abstract mathematical models, programming languages, and models of computer hardware.

6.2 The RAM Model

Our first attempt at a precise model of computation is the RAM model, which models memory as an infinite array M of natural numbers.

Definition 6.2. RAM program syntax
Suppose memory is an infinite array M of natural numbers. Then a RAM program $P = (V, C_0, \ldots, C_{\ell-1})$ consists of a finite set V of variables (or registers), and a sequence $C_0, C_1, \ldots, C_{\ell-1}$ of commands (or lines of code), each chosen from the following:

- (assignment to a constant) $\text{var} = c$, for a variable $\text{var} \in V$ and a constant $c \in \mathbb{N}$.
- (arithmetic) $\text{var}_0 = \text{var}_1$ op var_2, for variables $\text{var}_0, \text{var}_1, \text{var}_2 \in V$, and an operation op chosen from $+, -, \times, /$
- (read from memory) $\text{var}_0 = M[\text{var}_1]$ for variables $\text{var}_0, \text{var}_1 \in V$.
- (write to memory) $M[\text{var}_0] = \text{var}_1$ for variables $\text{var}_0, \text{var}_1 \in V$.
- (conditional goto) IF var $== 0$, GOTO k, where $k \in \{0, 1, \ldots, \ell\}$

In addition, we require that every RAM Program has three special variables: `input_len`, `output_ptr`, and `output_len`.

Definition 6.3. Computation of a RAM program
A RAM Program $P = (V, (C_0, \ldots, C_{\ell-1}))$ computes on an input x is as follows:

1. Initialization: The input x is encoded (in some predefined manner) as a sequence of natural numbers placed into memory locations $(M[0], \ldots, M[n-1])$, and all of the remaining memory locations are set to 0. The variable `input_len` is initialized to n, the length of x's encoding. All other variables are initialized to 0.

2. Execution: The sequence of commands $C_0, C_1, C_2, \ldots$ are executed in order (except when jumps are done due to GOTO commands), updating the values of variable and memory locations according to the usual interpretations of the operations. Since we are working with natural numbers, if the result of subtraction would be negative, it is replaced with 0. Similarly, the results of division are rounded down, and divide by 0 results in 0.

3. Output: If line ℓ is reached (possibly due to a GOTO ℓ), the output $P(x)$ is defined to be the subarray of M of length `output_len` starting at location `output_ptr`. That is,

$$P(x) = (M[\texttt{output\_ptr}], M[\texttt{output\_ptr} + 1], \ldots, M[\texttt{output\_ptr} + \texttt{output\_len} - 1])$$

The running time of P on input x, denoted $T_P(x)$, is defined to be: the number of commands executed during the computation (possibly ∞).

7 RAM Simulations

7.1 Expressiveness: Simulating High-Level Programs

Definition 7.1. Simulating
Simulating one computational model $\mathcal{M}$ by another computational model $\mathcal{N}$ is a very common technique in both the theory and practice of computer science. This means that for every program P in model $\mathcal{M}$, there is a program Q in model $\mathcal{N}$ that is equivalent to P. Here equivalence means that for every input x, $P(x)$ has the "same behavior" as $Q(x)$, meaning that one program halts iff the other halts, and if they do halt, they halt with the same output.

Theorem 7.2. Bridge between PLs and RAM programs
To bridge between high-level PLs and RAM programs, we have the following theorem.

1. Every Python program (and C program, Java program, OCaml program, etc.) can be simulated by a RAM Program.
2. Conversely, every RAM program can be simulated by a Python program (and C program, Java program, OCaml program, etc.).

7.2 Robustness and Technological Relevance of the RAM model

We made somewhat arbitrary choices about what operations to include or not include in the RAM Model, mainly with an eye to the mathematical simplicity criterion. However, it turns out that the choice of operations does not affect what can be computed too much. For example, consider extending the RAM model with the modulo operation.

Theorem 7.3. Mod-extended RAM model
Define the mod-extended RAM model to be like the RAM model, but where we also allow a mod (%) operation. Then every mod-extended RAM program P can be simulated by a standard RAM program Q. Moreover, on every input x, the runtime of Q on x is at most 3 times the runtime of P on x.

Proof. Suppose we have an operation in our extended RAM model of the form $\text{var}_0 = \text{var}_1 \,\%\text{var}_2$. Instead, introduce a new temp variable temp, and replace this line of code with:

$$\begin{aligned} \text{temp} &= \text{var}_1 \,/\, \text{var}_2 \\ \text{temp} &= \text{temp} * \text{var}_2 \\ \text{var}_0 &= \text{var}_1 - \text{temp} \end{aligned}$$

The constant-factor blow up of 3 can be absorbed in $O(\cdot)$ notation, so we have

$$T_Q(x) = O\left(T_P(x)\right)$$

as desired. Thus, the choice of whether or not to include the mod operation does not affect the asymptotic growth rate of the runtime. □

On the opposite end, one might worry that the RAM model is too powerful and makes problems seem easier to solve than is possible in practice, e.g., because real devices (CPUs) have only a fixed number of registers compared to the arbitrary constant number of registers allowed by RAM programs, or because the values stored in registers and memory are not arbitrary natural numbers but in fact liited by word length (often 64 bits). However, the below may be proven to show that this is not a concern.

Theorem 7.4. RAM programs only need a few variables
There is a fixed constant c such that every RAM Program P can be simulated by a RAM program P' that uses at most c variables. (Our proof will have $c \leq 8$ but is not optimized.) Moreover, for every input x,

$$T_{P'}(x) = O\left(T_P(x) + |P(x)|\right)$$

where $|P(x)|$ denotes the length of P's output on x, measured in memory locations.

8 The Word-RAM Model and Randomized Algorithms

8.1 The Word-RAM Model

As noted above, an unrealistic feature of the RAM Model as we've defined it is it allows an algorithm to access and do arithmetic on arbitrarily large integers in one time step. In practice, the numbers stored in the registers of CPUs are of a modestly bounded word length w, e.g. $w = 64$ bits.

Definition 8.1. Word RAM Model
The Word RAM Model is defined like the RAM Model except that it has a word length w and memory size S that are used as follows:

- Memory: array of length S, with entries in $\{0, 1, \ldots, 2^w - 1\}$. Reads and writes to memory locations larger than S have no effect. Initially $S = n$, the length of the input array x. Additional memory can be allocated via a MALLOC command, which increments S by 1.
- Variables: In addition to the usual variables in a RAM Program, there is a variable `word_len` that is initialized to equal the word length w.
- Output: if the program halts, the output is defined to be $(M[\texttt{output\_ptr}], M[\texttt{output\_ptr} + 1], \ldots, M[\min\{\texttt{output\_ptr} + \texttt{output\_len} - 1, S - 1\}])$. That is, portions of the output outside allocated memory are ignored.
- Operations: Addition and multiplication are redefined from RAM Model to return $2^w - 1$ if the result would be $\geq 2^w$.
- Crashing: A Word-RAM program P crashes on input x and word length w if any of the following occur:
  - One of the constants c in the assignment commands ($\texttt{var} = c$) in P is $\geq 2^w$.
  - $x[i] \geq 2^w$ for some $i \in [n]$
  - $S > 2^w$. (This can happen because $n > 2^w$, or if $2^w - n + 1$ MALLOC commands are executed.)

We denote the computation of a Word-RAM program on input x with word length w by $P[w](x)$. Note that $P[w](x)$ has one of three outcomes:

- halting with an output
- failing to halt, or
- crashing.

We define the runtime $T_{P[w]}(x)$ to be the number of commands executed until P either halts or crashes (so $T_{P[w]}(x) = \infty$ if $P[w](x)$ fails to halt).

Definition 8.2. Word-RAM program running time
We say that word-RAM program P solves computational problem $\Pi = (\mathcal{I}, \mathcal{O}, f)$ if the following holds for every input x,

- There is at least one word length $w \in \mathbb{N}$ such that $P[w](x)$ halts without crashing.
- For every word length $w \in \mathbb{N}$ such that $P[w](x)$ halts without crashing, the output $P[w](x)$ satisfies $P[w](x) \in f(x)$ if $f(x) \neq \emptyset$ and $P[w](x) = \perp$ if $f(x) = \emptyset$.

The running time of a word-RAM program P on an input x is defined to be

$$T_P(x) = \max_{w \in \mathbb{N}} T_{P[w]}(x).$$

Like in Lecture 2, we define the worst-case running time of P to be the function $T_P(n)$ that is the maximum of $T_P(x)$ over inputs x of size at most n.

In many algorithms texts, you'll see the word size constrained to be $O(\log n)$, where n is the length of the input. This is justified by the following:

Proposition 8.3. Existence of sufficiently large w to guarantee not crashing
For a word-RAM program P and an input x that is an array of numbers, if $T_P(x) < \infty$, then there is a word size w_0 such that $P[w](x)$ does not crash for any $w \geq w_0$. Specifically, we can take

$$w_0 = \lceil \log_2 \max\{n + T_P(x), x[0], \ldots, x[n-1], c_0, \ldots, c_{k-1}\} + 1 \rceil$$

where $c_0, \ldots, c_{k-1}$ are the constants appearing in variable assignments in P.

This proposition implies that if the runtime $T_P(x)$ is at most poly (n) and the numbers in x are of magnitude at most poly(n), the we need word size at most $w_0 = O(\log n)$ to support the computation. (The constants $c_0, \ldots, c_{k-1}$ are just constants and have no dependence on n.) Most algorithms courses focus on algorithms that run in time poly (n), so there is no harm in restricting to word length $O(\log n)$. However, in CS1200, we will sometimes study algorithms that run in exponential time or don't even halt, and thus may also use much more than polynomial (in n) memory locations.

Proof. The setting of w_0 and $w \geq w_0$ ensures that

$$2^w \geq 2^{w_0} \geq \max\{n + T_P(x), x[0], \ldots, x[n-1], c_0, \ldots, c_{k-1}\} + 1$$

In an execution of $P[w](x)$, the memory size S never exceeds $n + T_P(x)$, because it starts at n and grows by at most one in each time step (which happens only if a MALLOC command is executed). Thus we are guaranteed to maintain $S \leq 2^w - 1$, so that the program won't crash because of allocating too much memory. Since $x[0], \ldots, x[n-1] \leq 2^w - 1, P[w](x)$ will not crash because of values assigned to memory locations. Since $c_0, \ldots, c_{k-1} \leq 2^w - 1$, then $P[w](x)$ will not crash because of variable assignments. □

8.2 Randomized Algorithms

As we have currently constructed, programs in the RAM and Word-RAM models have deterministic output.

Definition 8.4. Extending RAM model to randomness
We can model randomization by adding to the RAM or Word-RAM Model a new random command, used as follows:

$$\text{var}_1 = \text{random}\,(\text{var}_2)$$

which assigns var_1 a uniformly element of the set $[\text{var}_2] \in \{0, 1, \ldots \text{var}_2 - 1\}$.

Definition 8.5. Randomized algorithms
There are two flavors of randomized algorithms.

- **Las Vegas Algorithms:** these are algorithms that always output a correct answer, but their running time depends on their random choices. For some very unlikely random choices, Las Vegas algorithms are allowed to have very large running time. Typically, we try to bound their expected running time. That is, we say the (worst-case) expected running time of A is

  $$T_{\text{expected}}\,(n) = \max_{x:\text{size}(x)\leq n} \text{E}\,[T_A(x)]$$

  where $T_A(x)$ is the random variable denoting the runtime of A on x, and $\text{E}[\cdot]$ denotes the expectation of a random variable:

  $$\text{E}[Z] = \sum_{z\in\mathbb{N}} z \cdot \Pr[Z = z]$$

- **Monte Carlo Algorithms:** these are algorithms that always run within a desired time bound $T(n)$, but may err with some small probability (if they are unlucky in their random choices), i.e. we say that A solves computational problem $\Pi = (\mathcal{I}, \mathcal{O}, f)$ with error probability p if the following holds for every $x \in \mathcal{I}$,

  1. $\Pr[A(x) \in f(x)] \geq 1 - p$ if $f(x) \neq \emptyset$ and
  2. $\Pr[A(x) = \perp] \geq 1 - p$ if $f(x) = \emptyset$.

  Typically the constant p can be reduced to an astronomically small value by running the algorithm several times independently.

It turns out that every Las Vegas algorithm can be converted into a Monte Carlo, but it is not known whether the converse holds. However, there are problems for which Monte Carlo algorithms achieve better runtime.

Theorem 8.6. QuickSelect
There is a randomized algorithm QuickSelect that always solves Selection correctly, and has (worst-case) expected running time $O(n)$. In particular,

Algorithm 3 QuickSelect

Input: An array $A = ((K_0, V_0), \ldots, (K_{n-1}, V_{n-1}))$, where each $K_j \in \mathbb{R}$, and $i \in \mathbb{N}$
Output: A key-value pair (K_j, V_j) such that K_j is an $i + 1^{\text{st}}$ smallest key

```
procedure QuickSelect(A, i)
    if n ≤ 1 then
        return (K_0, V_0)
    else
        p = random(n);
        Pivot = K_p;
        Let A_smaller = an array containing the elements of A with keys < Pivot;
        Let A_larger = an array containing the elements of A with keys > Pivot;
        Let A_equal = an array containing the elements of A with keys = Pivot;
        Let n_smaller, n_larger, n_equal be the lengths of A_smaller, A_larger, and A_equal (so n_smaller +
        n_larger + n_equal = n);
        if i < n_smaller then
            return QuickSelect(A_smaller, i);
        else if i ≥ n_smaller + n_equal then
            return QuickSelect(A_larger, i − n_smaller − n_equal);
        else
            return A_equal[0];
```

9 Dictionaries

9.1 Randomized Data Structures

We can allow data structures to be randomized, by allowing the algorithms Preprocess, EvalQ, and EvalU to be randomized algorithms, and again the data structures can either be Las Vegas (never make an error, but have random runtimes) or Monte Carlo (have fixed runtime, but can err with small probability).

Dynamic Dictionaries

Definition 9.1. Dynamic dictionaries
The dynamic dictionaries data-structure problem is as follows:

- **Updates:** Insert or delete a key-value pair (K, V) with $K \in [U]$ into the multiset
- **Queries:** Search - given a key K, return a matching key-value pair (K, V) from the multiset

Proposition 9.2. Dynamic dictionaries solution
Dynamic Dictionaries has a solution in which Insert, Delete and Search can be achieved in $O(1)$ time and Preprocessing involves initializing an empty memory of size $O(U)$ and additional $O(1)$ time.

Proof. The algorithm is as follows:

- Preprocess: initialize an array A of length U, with each entry being the start of an empty linked list.
- Insert: for a key-value pair (K, V), add (K, V) as the head of the linked list starting at $A(K)$.
- Delete: for a key K, delete the head of the linked list starting at $A(K)$ (if it exists).
- Search: for a key K, return the key-value pair at the head of the linked list starting at $A(K)$, if it exists. If not, return $\perp$.

This is essentially what we saw in `SingletonBucketSort()`. □

Definition 9.3. Random hash functions
These are a collection of functions $h_\ell : [U] \to [m]$, where $\ell \in [R]$ for some number R that can be large, with the following properties:

- Efficient evaluation: Given $\ell \in [R]$ and $K \in [U]$, evaluating $h_\ell(K)$ takes time $O(1)$.
- Low chances of collision: For any two $K, K' \in [U]$, the fraction of ℓ such that $h_\ell(K) = h_\ell(K')$ is $O(1/m)$. That is,

$$\Pr_\ell \left[h_\ell(K) = h_\ell\left(K'\right) \right] = O\left(\frac{1}{m}\right)$$

  when ℓ is chosen uniformly at random from $[R]$.

Theorem 9.4. Dynamic dictionaries via hash tables
For each positive integer m, Dynamic Dictionaries has a Las Vegas randomized solution in which Insert, Delete, and Search can be achieved in $O\left(1+\frac{n}{m}\right)$ expected time and Preprocessing involves initializing an empty memory of size $O(m)$ along with additional $O(1)$ runtime. Here, n is the maximum number of key-value pairs in the data structure till the time under consideration.

Whenever $m = \Theta(n)$, the runtime of updates and queries is $O(1)$. If n becomes larger, one has to update the data structure by choosing a larger value of m.

Proof. We define operations as follows.

- Preprocess: initialize an array A of length m, with each entry being the start of an empty linked list. Choose and store a random $\ell = \text{random}(R)$, indexing a hash function h_ℓ from our collection.
- Insert: for a key-value pair (K, V), add (K, V) as the head of the linked list starting at $A\left(h_\ell(K)\right)$.
- Delete: for a key K, consider the linked list starting at $A\left(h_\ell(K)\right)$. Traverse this linked list from the head and delete the first key-value pair with key K (if it exists).
- Search: for a key K, consider the linked list starting at $A\left(h_\ell(K)\right)$. Traverse this linked list from the head and return the first key-value pair with key K, if it exists. If no key-value pair with key K is found, return $\perp$.

The algorithm correctly returns the search queries by construction, and hence it is a Las-Vegas data structure. The preprocessing step initializes an empty memory of size $O(m)$ and takes $O(1)$ step for choosing the random number. The runtime of insert operation is the time taken to evaluate $h_\ell(K)$ and add (K, V) at the start of the linked list, both of which is $O(1)$.

The runtime of delete and search operations is dominated by the time taken to traverse the linked list at $A\left(h_\ell(K)\right)$ up until the key K is found (as above, other operations such as evaluating $h_\ell(K)$ is $O(1)$ time). This time is $O\left(\text{length}_{\ell}(K)\right)$, where length (K) is the length of the linked list from the head to the first occurrence of key K at the location $h_\ell(K)$, under ℓ-th hash function. If we can show that the expected length

$$\mathrm{E}_\ell\left[\text{length}_\ell(K)\right] = O\left(1+\frac{n}{m}\right),$$

the expected runtime bound concludes. This can be shown by noting that the probability that two keys go to the same index is $1/m$, such that the expected number of keys different from K that are in the same linked list as K is $O(n/m)$. This shows that the number of key-value pairs starting from $A\left(h_l(K)\right)$ to K is $O(1+n/m)$. (A formal proof appears in Section 3.4 of the notes, but it is an optional read.) $\square$

Remark 9.5. Load of hash table
We call $\alpha = n/m$ the load of the data structure, so the expected runtime of the hash map is $O(1 + \alpha)$. Notice that we get $O(1)$ expected time as long as $m = \Omega(n)$. To maintain the time efficiency, we need to tailor m to the size n of the dataset, which we may not know advance. This can be solved by rebuilding the data structure as the hash table gets too full. For example, when we reach load $\alpha = 2/3$, we can increase m by a factor of 2 (and use new hash functions) to bring us back to $\alpha = 1/3$. The cost associated to the rebuilding of data structure is proportional to the number of key-value pairs that it took to increase the load from 1/3 to 2/3, and hence the average cost per key-value pair (referred to as the amortized cost) is $O(1)$.

Storing and Searching Data Synthesis

Remark 9.6. Summary of storing and searching key-value pairs
We summarize our approaches thus far for storing and searching datasets of key-value pairs:

- **Sort the dataset and store the sorted array:** Selection queries can be done in $O(1)$ (after preprocessing of $O(n \log n)$ to initially sort the array).
- **Store in a binary search tree (balanced and appropriately augmented):** A BST is a dynamic data structure which allows insert, delete, predecessor, and selection all in $O(h) = O(\log n)$ time.
- **Run Randomized QuickSelect:** One selection query takes $O(n)$, which is quicker than the time to preprocess/insert the entire dataset of the above options.
- **Store in a hash table:** Search and updates can be done in $O(1)$ time, as long as it's not very overloaded.

10 Graph Search

10.1 Graph Algorithms

Definition 10.1. Graph
A graph $G = (V, E)$ consists of a finite set of vertices V (sometime called nodes), and a set E of edges, which are ordered pairs (u, v) where $u, v \in V$ and $u \neq v$.

Unless we state otherwise, assume:

- graph means a simple, unweighted, undirected graph;
- digraph means a simple, unweighted, directed graph.

Of course, we need to specify how the graph is encoded as an input to our algorithm. In CS 1200, we work exclusively with adjacency lists.

Definition 10.2. Adjacency lists
We will work with the adjacency list representation, where for each vertex v, we are given an out-neighbor array $\texttt{Nbr}_{\text{out}}[v]$ storing the elements of the set

$$\mathrm{N}_{\text{out}}(v) = \{u : (v, u) \in E\}.$$

The array $\texttt{Nbr}_{\text{out}}[v]$ is given together with its length, which is equal to

$$\deg_{\text{out}}(v) = |\mathrm{N}_{\text{out}}[v]|\,.$$

Thus, for $i \in [\deg_{\text{out}}(v)]\,, \texttt{Nbr}_{\text{out}}[v][i] = u_i$ where $u_0, \ldots, u_{\deg_{\text{out}}(v)-1}$ is an arbitrary ordering of the elements of $\mathrm{N}_{\text{out}}(v)$.

Analogously, the in-neighbors are $\mathrm{N}_{\text{in}}(v) = |\{u : (u, v) \in E\}|$ and the in-degree is $\deg_{\text{in}}(v) = |\mathrm{N}_{\text{in}}(v)|$, but we won't have those given explicitly as input to our algorithms. For undirected graphs, we will just write $N(v) = \mathrm{N}_{\text{in}}(v) = \mathrm{N}_{\text{out}}(v)$ for the neighbors of v and $\deg(v) = \deg_{\text{in}}(v) = \deg_{\text{out}}(v)$ for the degree of v.

10.2 Shortest Walks

Definition 10.3. Walks
Let $G = (V, E)$ be a directed graph, and $s, t \in V$.

- A walk w from s to t in G is a sequence $v_0, v_1, \ldots, v_\ell$ of vertices such that $v_0 = s, v_\ell = t$, and $(v_{i-1}, v_i) \in E$ for $i = 1, \ldots, \ell$.
- A walk in which all vertices are distinct is also called a path.
- The length of a walk w is length (w) = the number of edges in w (the number ℓ above).
- The distance from s to t in G is

$$\text{dist}_G(s,t) = \begin{cases} \min\{\text{length}(w) : w \text{ is a walk from } s \text{ to } t\} & \text{if a walk exists} \\ \infty & \text{otherwise} \end{cases}$$

- A shortest walk from s to t in G is a walk w from s to t with $\text{length}(w) = \text{dist}_G(s,t)$

Lemma 10.4. Shortest walk is a path
If w is a shortest walk from s to t, then all of the vertices that occur on w are distinct (i.e. w is a path).

Proof. Suppose for contradiction that there is a shortest walk $w = (s = v_0, v_1, \ldots, v_\ell = t)$ that does not satisfy this property, i.e. $v_i = v_j$ for some $i < j$. But then we can cut out the loop $(v_i, v_{i+1}, \ldots, v_j)$ and produce the walk $w' = (s = v_0, \ldots, v_{i-1}, v_i = v_j, v_{j+1}, \ldots, v_\ell)$. We have the length of w' is strictly less than that of w and has the same start and endpoints. But then w is not a shortest walk, so we have a contradiction. □

10.3 Breadth-First Search

Algorithm 4 Breadth-First Search

Input: A digraph $G = (V, E)$ and a source vertex $s \in V$
Output: The array $\texttt{dist}_s[\cdot] = \texttt{dist}_G(s, \cdot)$

procedure BFS(G, s)
    Initialize $\texttt{dist}[t] = \infty$ for all $t \in V$
    $S = F = \{s\}$
    $\texttt{dist}_s[s] = 0$
    for $d = 1, \ldots, n-1$ **do**
        Let $F = \{v \in V : v \notin S, \exists u \in F \text{ s.t. } (u, v) \in E\}$
        For every $v \in F$, $\texttt{dist}_s[v] = d$
        $S = S \cup F$
return $\texttt{dist}_s$

Proof. We induct on d to show that

$$F_d = \{v \in V : \text{dist}_G(s, v) = d\}.$$

and

$$S_d = \{v \in V : \text{dist}_G(s, v) \leq d\}.$$

Once proven, this implies that in iteration d we correctly set $\texttt{dist}[v] = d$ for every vertex v at distance d. Since all vertices either have distance at most $n-1$ or ∞ from s, the output array `dist`, is correct. □

Theorem 10.5. BFS runtime
The algorithm $\text{BFS}(G)$ correctly solves SingleSourceDistances and can be implemented in time $O(n+m)$, where n is the number of vertices in G and m is the number of edges in G.

Proof. On implementation details: in practice, F is maintained as a queue and vertices u are processed one at time. Initially the queue starts with vertex s. As long as the queue is nonempty, we remove the vertex u at the head of the queue, go over the vertices v in its adjacency list $\texttt{Nbr}_{\text{out}}[u]$, and for each one where $v \notin S$, we add v to both S and the end of the queue and set $\texttt{dist}_s[v] = \texttt{dist}_s[u] + 1$. This algorithm is equivalent to Algorithm 1, with a particular order of processing vetices. □

10.4 Finding the Paths

Theorem 10.6. Solving SingleSourceShortestPaths
The data-structure problem SingleSourceShortestPaths takes as input a digraph $G = (V, E)$ and a source vertex $s \in V$, and may be queried for any query vertex t and return a shortest path from s to t (if one exists).

There is a solution that solves the SingleSourceShortestPaths problem on digraphs with n vertices and m edges with preprocessing time $O(n+m)$ and the time to answer a query t is $O(\text{dist}_G(s, t))$.

Proof. We can augment BFS to maintain an auxiliary array A_{pred} of size $|V|$, where A_{pred} holds the vertex u that we "discovered" v from. That is, if we add v to F because of the edge (u, v) (which would happen while going over the adjacency list of u), we set $A_{\text{pred}}[v] = u$. After the completion of BFS, we can reconstruct the path from s to t using this predecessor array, namely as $t, A_{\text{pred}}[t], A_{\text{pred}}[A_{\text{pred}}[t]], \ldots$ until we reach s. □

Corollary 10.7. Solving ShortestWalks with BFS
ShortestWalks (equivalently, ShortestPaths) can be solved in time $O(n+m)$.

11 Graph Coloring

11.1 Motivating graph coloring

The computational problem of finding a coloring is called the Graph Coloring problem.

Definition 11.1. *k-coloring*
For an undirected graph $G = (V, E)$, a proper k-coloring of G is a mapping $f : V \to [k]$ such that for all edges $\{u, v\} \in E$, we have $f(u) \neq f(v)$.

11.2 Greedy Coloring

A first attempt at graph coloring is the greedy strategy.

Algorithm 5 Greedy Coloring

Input: A graph $G = (V, E)$
Output: A coloring f of G using "few" colors

procedure GreedyColoring(G)
    Select an ordering $v_0, v_1, v_2, \ldots, v_{n-1}$ of V
    for $i = 0, \ldots, n-1$ **do**
        $f(v_i) = \min\{c \in \mathbb{N} : c \neq f(v_j) \, \forall j < i \text{ s.t. } \{v_i, v_j\} \in E\}$.
    return f

Theorem 11.2. Bounding the number of colors from greedy coloring
When run on a graph $G = (V, E)$ with any ordering of vertices, GreedyColoring (G) will use at most $\deg_{\max} + 1$ colors, where $\deg_{\max} = \max\{d(v) : v \in V\}$.

Proof. The set $\{f(v_j) : \exists j < i \text{ s.t. } \{v_i, v_j\} \in E\}$ has size at most $d(v_j) \leq \deg_{\max}$, so cannot include all of the colors $0, 1, 2, \ldots,$ deg $_{\max}$. Thus when we assign $f(v_i)$ to be the minimum element outside that set, we will have $f(v_i) \in [\deg_{\max} + 1]$. □

The performance of greedy algorithms can be very sensitive to the order in which decisions are made, and often we can achieve much better performance by picking a careful ordering.

Algorithm 11.3. BFSColoring (modified Greedy Coloring)
If we fix a random $v_0 \in V$ and then perform BFS, we may obtain a list of vertices in BFS order. If we then perform GreedyColoring over this ordering of vertices, then we obtain the algorithm BFSColoring.

Theorem 11.4. BFSColoring solves 2-coloring
If G is a connected 2-colorable graph, then BFSColoring(G) will color G using 2 colors.

12 Interval Scheduling, Independent Sets, and Matching

12.1 Interval Scheduling

Algorithm 12.1. IntervalSchedulingOptimization via GreedyIntervalScheduling
The IntervalSchedulingOptimization problem is as follows: given a list x of n intervals, we wish to output a "large" subset of the input intevals that are disjoint from each other.

The GreedyIntervalScheduling algorithm solves this as follows: sort the list of intervals by increasing end time, initialize a schedule set S as the empty set, and then iterate through the array and add to S each element as long as it does not conflict with the existing elements.

It may be shown that GreedyIntervalScheduling(x) finds an optimal solution to IntervalSchedulingOptimization, and can be implemented in time $O(n \log n)$.

12.2 Independent Set

By considering each interval as a node and letting each pair of conflicting intervals to be connected by an edge, we may generalize the interval scheduling problem to the Independent Set problem.

Definition 12.2. Independent set
Let $G = (V, E)$ be a graph. An independent set in G is a subset $S \subseteq V$ such that there are no edges entirely in S. That is, $\{u, v\} \in E$ implies that $u \notin S$ or $v \notin S$.

There are no known efficient algorithms for the Independent Set problem, but a greedy algorithm may be designed to output a somewhat large independent set.

Algorithm 6 GreedyIndSet (greedy algorithm to find independent sets on a graph)

Input: A graph $G = (V, E)$
Output: A "large" independent set in G

procedure GreedyIndSet(G)
    Choose an ordering $v_0, v_1, v_2, \ldots, v_{n-1}$ of V;
    $S = \emptyset$;
    for $i = 0, \ldots, n-1$ **do**
        if $v_j \notin S \quad \forall j < i$ s.t. $\{v_i, v_j\} \in E$ **then**
            $S = S \cup v_i$;
    return S

Theorem 12.3. Runtime of GreedyIndSet
For every graph G with n vertices and m edges, GreedyIndSet(G) can be implemented in time $O(n + m)$ and outputs an independent set of size at least $n/\left(\deg_{\max} + 1\right)$, where $\deg_{\max}$ is the maximum vertex degree in G.

Proof. We prove this via contradiction. Suppose the size of the independent set output by the algorithm is $K < n/\left(\deg_{\max} + 1\right)$. The number of vertices either in this independent set or having

an edge with a vertex in this independent set are $K(\deg_{\max}+1) < n$, thus at least one vertex denoted v_ℓ—does not have this property. Considering the execution of Line 5 for v_ℓ, we see that it must have been added to the independent set, leading to size $K+1$, which is a contradiction. □

12.3 Matching

Definition 12.4. Matching
For a graph $G = (V, E)$, a matching in G is a subset $M \subseteq E$ such that every vertex $v \in V$ is incident to at most one edge in M.Equivalently, no two edges in M share an endpoint.

Definition 12.5. Maximum Matching
The problem of finding the largest matching in a graph is called Maximum Matching. That is, given input a graph $G = (V, E)$, we wish to output a matching $M \subseteq E$ in G of maximum size.

Definition 12.6. Alternating walk, augmenting path
Let $G = (V, E)$ be a graph, and M be a matching in G. Then:

- An alternating walk W in G with respect to M is a walk $(v_0, v_1, \ldots, v_\ell)$ in G such that for every $i = 1, \ldots, \ell - 1$, exactly one of $\{v_{i-1}, v_i\}$ and $\{v_i, v_{i+1}\}$ is in M.
- An augmenting path P in G with respect to M is an alternating walk in which v_0 and v_ℓ are respectively unmatched by M, and in which all of the vertices in the walk are distinct, and $\ell \geq 1$.

This suggests a natural algorithm for maximum matching: repeatedly try to find an augmenting path and use it to grow our matching. But we need to argue that augmenting paths always exist and we can find them efficiently. An important piece in this argument is the following theorem.

Theorem 12.7. Berge's Theorem
Let $G = (V, E)$ be a graph, and $M \subseteq E$ be a matching. If (and only if) M is not a maximum-size matching, then G has an augmenting path with respect to M.

Definition 12.8. Bipartite graph
A graph $G = (V, E)$ is bipartite if it is 2-colorable. That is, there is a partition of vertices $V = V_0 \cup V_1$ (with $V_0 \cap V_1 = \emptyset$) such that all edges in E have one endpoint in V_0 and one endpoint in V_1.

13 Matchings

13.1 Maximum Matching Algorithm

Like in a greedy strategy, we will try to grow our matching M on step at a time, building a sequence $M_0 = \emptyset, M_1, M_2, \ldots$, with $|M_k| = k$. However, to get M_k from M_{k-1} we will sometimes do more sophisticated operations than just adding an edge. Instead we will rely on Berge's Theorem, which tell us that if our matching is not of maximum size, then there is an augmenting path. We will obtain an algorithm by the following two lemmas.

Lemma 13.1. Constructing a matching from an augmenting path
Given a graph $G = (V, E)$, a matching M, and an augmenting path P with respect to M, we can construct a matching M' with $|M'| = |M| + 1$ in time $O(n)$.

Proof. Let $P = (v_0, \ldots, v_\ell)$ be an augmenting path. We have that $\{v_0, v_1\} \in E \backslash M$ since v_0 is unmatched, and $\{v_{\ell-1}, v_\ell\} \in E \backslash M$ since v_ℓ is unmatched. Thus, let

$$M' = (M - \{\{v_1, v_2\}, \{v_3, v_4\}, \ldots, \{v_{\ell-2}, v_{\ell-1}\}\}) \cup \{\{v_0, v_1\}, \{v_2, v_3\} \ldots, \{v_{\ell-1}, v_\ell\}\}.$$

In words, we flip each of the edges in this augmenting path: any edge of the augmenting path originally in the matching M is now not in the matching M', and any edge of the augmenting path not in M is in M'.

To show M' is a matching, we need to argue that every vertex in G is incident to at most one edge from M'. None of the edges we have added to the matching M are incident any vertices other than $v_0, \ldots, v_\ell$, so we only need to check that the matching property holds for those vertices. Now v_0 and v_ℓ were previously unmatched, and they are each incident to exactly one of the new edges we have added from P (since paths have no repeated vertices by definition). The vertices $v_1, \ldots, v_{\ell-1}$ were each previously incident to exactly one edge of the matching M, which we have removed, and they each are incident to exactly one of the new edges we have added.

M' matches two more vertices than M, namely v_0 and v_ℓ, so $|M'| = |M| + 1$. □

Lemma 13.2. Finding augmenting paths for matchings on bipartite graphs
Given a bipartite graph $G = (V, E)$ and a matching M that is not of maximum size, we can find an augmenting path with respect to M in time $O(n + m)$.

Proof. To see that this is true, observe the following lemma. □

Lemma 13.3. Finding shortest alternating walks in bipartite graphs reduces to finding shortest paths in directed graphs
Let $G = (V_0 \cup V_1, E)$ be bipartite and let M be a matching in G that is not of maximum size. Let U be the vertices that are not matched by M, and $U_0 = V_0 \cap U$ and $U_1 = V_1 \cap U$. Then

- G has an alternating walk with respect to M that starts in U_0 and ends in U_1.
- Every shortest alternating walk from U_0 to U_1 is an augmenting path.

From this, we may derive a reduction: finding shortest alternating walks in bipartite graphs reduces to finding shortest paths in directed graphs in time $O(n + m)$.

Proof. As a sketch of the reduction: Given G, we construct a directed graph G' as follows:

- Direct edges in $E \backslash M$ to go from V_0 to V_1,
- Direct edges in M to go from V_1 to V_0,
- Add a source vertex s with all edges going from s to U_0,
- Add a target vertex t with all edges going from U_1 to t.

Then walks of length ℓ from s to t in G' correspond to alternating walks of length $\ell - 2$ in G from U_0 to U_1 (just drop s and t from the walk), and thus shortest walks from s to t correspond to shortest alternating walks from U_0 to U_1. □

Since a matching can be of size at most $n/2$, repeatedly applying the first two lemmas gives us an algorithm that runs in time $(n/2) \cdot (O(n) + O(n+m)) = O(n \cdot (n+m))$. Moreover, by eliminating isolated vertices, in time $O(n)$ we can reduce to the case where $n \leq 2m$, giving us a run time of $O(n) + O(n \cdot (2m + m)) = O(nm)$. Thus we have:

Theorem 13.4. Maximum Matching on bipartite graphs
Maximum Matching can be solved in time $O(mn)$ on bipartite graphs with m edges and n vertices.

Algorithm 7 Max Matching on Bipartite Graphs

Input: A bipartite graph $G = (V, E)$
Output: A maximum-size matching $M \subseteq E$

```
procedure MaxMatchingAugPaths(()G)
    Remove isolated vertices from G
    Let V_0, V_1 be the bipartition (i.e. 2-coloring) of V
    M = ∅;
    while P is not ⊥ do
        Let U be the vertices unmatched by M, U_0 = V_0 ∩ U, U_1 = V_1 ∩ U
        Use BFS to find a shortest alternating walk P that starts in U_0 and ends in U_1
        If P ≠⊥ then augment M using P via Lemma 5.1
    return M
```

14 Embedded EthiCS

Omitted.

15 Logic

15.1 Propositional Logic

Definition 15.1. Boolean formulas, informal
A boolean formula φ is a formula built up from a finite set of variables, say $x_0, \ldots, x_{n-1}$, using the logical operators $\wedge$ (AND), $\vee$(OR), and $\neg$ (NOT), and parentheses. Every boolean formula φ on n variables defines a boolean function, which we'll abuse notation and also denote by $\varphi : \{0,1\}^n \to \{0,1\}$, where we interpret 0 as false and 1 as true.

As stated in the following lemma, all boolean functions are expressible as both DNF and CNF formulas.

Definition 15.2. DNF and CNF formulas
We begin by defining literals, from which we build sequences of literals.

- A literal is a variable (e.g. x_i) or its negation ($\neg x_i$).
- A term is an AND of a sequence of literals.
- A clause is an OR of a sequence of literals.
- A boolean formula is in disjunctive normal form (DNF) if it is the OR of a sequence of terms.
- A boolean formula is in conjunctive normal form (CNF) if it is the AND of a sequence of clauses.

By convention, an empty term is always true and an empty clause is always false.

Remark 15.3. Simplifying clauses
Note terms and clauses may contain duplicate literals, but if a term or clause contains multiple copies of a variable x, it's equivalent to a term or clause with just one copy (since $x \vee x = x$ and $x \wedge x = x$). We can also remove any clause or term with both a variable x and its negation $-x$, as that clause or term will be always true (in the case of a clause) or always false (in the case of a term). We define a function Simplify which takes a clause and performs those simplifications: that is, given a clause B, Simplify(B) removes duplicates of literals from clause B, and returns 1 if B contains both a literal and its negation. Also, if we have an order on variables (e.g. $x_0, x_1, \ldots$), Simplify(B) also sorts the literals in order of their variables.

Lemma 15.4. All booleans are expressible as both DNF and CNF
For every boolean function $f : \{0,1\}^n \to \{0,1\}$, there are boolean formulas φ and ψ in DNF and CNF, respectively, such that $f \equiv \varphi$ and $f \equiv \psi$, where we use $\equiv$ to indicate equivalence as functions, i.e. $f \equiv g$ iff $\forall x : f(x) = g(x)$.

Proof. For a function $f : \{0,1\}^n \to \{0,1\}$, we can define the DNF

$$\varphi(x_0, \ldots, x_{n-1}) = \bigvee_{\alpha \in \{0,1\}^n : f(\alpha)=1} ((x_0 = \alpha_0) \wedge (x_1 = \alpha_1) \wedge \cdots \wedge (x_{n-1} = \alpha_{n-1})).$$

Note that $x_i = \alpha_i$ can be rewritten as either x_i (with $\alpha_i = 1$) or $\neg x_i$ (with $\alpha_i = 0$). For example, applying to the palindrome function on 4 bits , we get

$$\varphi(x_0, x_1, x_2, x_3) = (x_0 \wedge x_1 \wedge x_2 \wedge x_3) \vee (x_0 \wedge \neg x_1 \wedge \neg x_2 \wedge x_3) \vee (\neg x_0 \wedge x_1 \wedge x_2 \wedge \neg x_3) \vee (\neg x_0 \wedge \neg x_1 \wedge \neg x_2 \wedge \neg x_3),$$

where the terms correspond to the satisfying assignments $(1,1,1,1), (1,0,0,1), (0,1,1,0)$, and $(0,0,0,0)$.

For the CNF, we define

$$\varphi(x_0, \ldots, x_{n-1}) = \bigwedge_{\alpha \in \{0,1\}^n : f(\alpha)=0} ((x_0 \neq \alpha_0) \vee (x_1 \neq \alpha_1) \vee \cdots \vee (x_{n-1} \neq \alpha_{n-1})).$$

□

15.2 Computational Problems in Propositional Logic

Definition 15.5. Propositional logic problems
From these definitions, the following problems arise naturally:

- **Satisfiability**
  - Input: A boolean formula φ on n variables
  - Output: An $\alpha \in \{0,1\}^n$ such that $\varphi(\alpha) = 1$ (if one exists)
- **CNF-Satisfiability**
  - Input: A CNF formula φ on n variables
  - Output: An $\alpha \in \{0,1\}^n$ such that $\varphi(\alpha) = 1$ (if one exists)
- **DNF-Satisfiability**
  - Input: A DNF formula φ on n variables
  - Output: An $\alpha \in \{0,1\}^n$ such that $\varphi(\alpha) = 1$ (if one exists)

Remark 15.6. DNF-Sat problems are easy, but the reduction from Sat to DNF-Sat is not polynomial unlike CNF-Sat
DNF satisfiability is easy—since we only need to satisfy a single term and then we are done, the only cases in which a DNF is unsatisfiable are when we have zero terms or every term has a contradiction (both a variable and its negation). On the other hand, it is known that Satisfiability can be reduced to CNF-satisfiability, so when we say "SAT," we refer to CNF-Satisfiability by default whenever it's convenient.

15.3 Modelling Using Satisfiability

One of the reasons for the importance of Satisfiability is its richness in encoding other problems. Thus any effort gone into optimizing SAT Solvers (i.e., algorithms for CNF-Satisfiability) can be easily be applied to other problems we want to solve. For example:

Theorem 15.7. Graph-
Graph k-Coloring on graphs with n nodes and m edges can be reduced in time $O(n + km)$ to (CNF-)Satisfiability with kn variables and $n + km$ clauses.

Proof. Given $G = (V, E)$ and $k \in \mathbb{N}$, we will construct a CNF ϕ_G that captures the graph k-coloring problem. In ϕ_G, we introduce indicator variables $x_{v,i}$ where $v \in V$ and $i \in [k]$, which intuitively are meant to correspond to vertex v being assigned to color i.

We then have a few types of clauses:

1. $(x_{v,0} \vee x_{v,1} \vee \cdots \vee x_{v,k-1})$ for all $v \in V$. (Each vertex must be assigned a color)
2. $(\neg x_{u,i} \vee \neg x_{v,i})$ for every edge $\{u, v\} \in E$ and every color $i \in [k]$. From De Morgan's Law, this is equivalent to the more intuitive $\neg (x_{u,i} \wedge x_{v,i})$. (The endpoints of an edge cannot be assigned the same color.)
3. In addition, we can require all vertices to be assigned at most one color. (The constraints above allow a satisfying assignment to assign multiple colors to the same vertex.) To avoid this, we can include clauses $(\neg x_{v,i} \vee \neg x_{v,j})$ for $i, j \in [k]$ where $i \neq j$. But for the case of coloring, we don't actually need to include these, and doing so would increase the number of clauses to nk^2.

We then call the SAT oracle on ϕ_G and get an assignment α. If $\alpha = \perp$, we say G is not k-colorable. Otherwise, we construct and output the coloring f_α given by:

$$f_\alpha(v) = \min \{i \in [k] : \alpha_{v,i} = 1\}.$$

□

16 Resolution

Definition 16.1. CNF formulas as a set of clauses
From now on, it will be useful to view a CNF formula as just a set $\mathcal{C}$ of clauses. Let $\mathcal{C}$ be a set of clauses over variables $x_0, \ldots, x_{n-1}$. We say that an assignment $\alpha \in \{0,1\}^n$ satisfies $\mathcal{C}$ if α satisfies all of the clauses in $\mathcal{C}$, or equivalently α satisfies the CNF formula

$$\varphi(x_0, \ldots, x_{n-1}) = \bigwedge_{C \in \mathcal{C}} C(x_0, \ldots, x_{n-1}).$$

SAT Solvers (i.e., algorithms to solve CNF-Satisfiability) have worst-case exponential running time, but on many realistic instances will terminate more quickly (with either a satisfying assignment, or an argument that the input formula is unsatisfiable). The best known SAT solvers implicitly use the technique of resolution, the idea of repeatedly deriving new clauses from the original clauses (using a valid deduction rule) until we either derive an empty clause (which is false, and thus we have a proof that the original formula is unsatisfiable) or we cannot derive any more clauses (in which case we can efficiently construct a satisfying assignment).

Definition 16.2. Resolution rule
For clauses C and D, define their resolvent to be

$$C \diamond D = \begin{cases} \text{Simplify}((C - \{\ell\}) \vee (D - \{\neg\ell\})) & \text{if } \ell \text{ is a literal s.t. } \ell \in C \text{ and } \neg\ell \in D \\ 1 & \text{if there is no such literal } \ell \end{cases}$$

Here $C - \{\ell\}$ means remove literal ℓ from clause C, and 1 represents true. Some remarks:

- As noted last time, if C and D can be resolved with respect to more than one literal ℓ, then for all choices of ℓ we will have $\text{Simplify}((C - \{\ell\}) \vee (D - \{\ell\})) = 1$, so $C \diamond D$ is well-defined.
- In the special case where $C = \ell, D = \neg\ell$, we use our definition from Lecture 15 that empty clause is always false and obtain

$$(\ell) \diamond (\neg\ell) = \emptyset = \text{FALSE}.$$

Theorem 16.3. Resolution Theorem
Let $\mathcal{C}$ be a set of clauses over n variables $x_0, \ldots, x_{n-1}$. Suppose that $\mathcal{C}$ is closed under resolution, meaning that for every $C, D \in \mathcal{C}$, we have $C \diamond D \in \mathcal{C} \cup \{1\}$. Then letting k be the maximum width over all clauses in $\mathcal{C}$, we have:

- $\emptyset \in \mathcal{C}$ iff $\mathcal{C}$ is unsatisfiable.
- If $\emptyset \notin \mathcal{C}$, then `ExtractAssignment`$(\mathcal{C})$ finds a satisfying assignment to $\mathcal{C}$ in time $O(n + k|\mathcal{C}|)$, where `ExtractAssignment`() is an algorithm described below and k is the maximum width over the clauses in $\mathcal{C}$.

Algorithm 16.4. `ResolutionInOrder` **and** `ExtractAssignment`($\mathcal{C}$)
To solve CNF-Satisfiability, our algorithm `ResolutionInOrder` starts with a set $\mathcal{C}$ of clauses and keep adding resolvents until we cannot add any new ones. At this point, we know that the new set of clauses is closed under resolution, and then (via the previous theorem) return unsatisfiable if $\emptyset$ is a clause and otherwise use `ExtractAssignment`() to find a satisfying assignment. Explicitly,

1. Resolve C_0 with each of $C_1, \ldots, C_{m-1}$, adding any new clauses obtained from the resolution $C_m, C_{m+1}, \ldots$. If $\emptyset$ clause is found, return unsatisfiable.
2. Resolve C_1 with each of $C_2, \ldots, C_{m-1}$ as well as with all of the resolvents obtained in Step 1, again adding any new clauses. If $\emptyset$ clause is found, return unsatisfiable.
3. Resolve C_2 with each of $C_3, \ldots, C_{m-1}$ as well as with all of the resolvents obtained in Steps 1 and 2 , again adding any new clauses. If $\emptyset$ clause is found, return unsatisfiable.
4. Continue on until we cannot add any more clauses.
5. Run `ExtractAssignment`() on the set of all clauses and return the satisfying assignment.

To run `ExtractAssignment`(), we take our updated set $\mathcal{C}$ of clauses which is closed under resolution (and doesn't contain $\emptyset$) and then generate a satisfying assignment one variable at a time in the following manner:

1. If $\mathcal{C}$ contains a singleton clause (v), then we assign $v = 1$.
2. If it contains $(\neg v)$ then assign $v = 0$.
3. If it contains neither (v) nor $(\neg v)$, then assign v arbitrarily.
4. $\mathcal{C}$ cannot contain both (v) and $(\neg v)$, because $\mathcal{C}$ is closed and does not contain 0.

17 Introduction to Limits of Computation, Church-Turing Thesis

For the remainder of this course, we discuss what algorithms cannot do efficiently and cannot do at all. Our main approach for doing so will be discussing some hard problems and using reductions.

Our discussion will focus on Word-RAM programs and the Word-RAM model, so we begin our exploration by discussing the Church-Turing Thesis, which guarantees that our upcoming results on computational limitations are agnostic to our choice of computational model (in the sense that choosing a different model would not achieve any more than a polynomial speedup).

Theorem 17.1. Turing-equivalent models
If a computational problem Π is solvable in one of the following models of computation, then it is solvable in all of them:

- RAM programs
- Word-RAM programs
- XOR-extended RAM or Word-RAM programs
- %-extended RAM or Word-RAM programs
- Python programs
- OCaml programs
- C programs (modified to allow a variable/growing pointer size)
- Turing machines
- Lambda calculus

More precisely, the above models of computation are equivalent up to the representation of input and output. Moreover, there is an algorithm (e.g. a RAM program) that can transform a program in any of these models of computation into an equivalent program in any of the others.

Theorem 17.2. Church-Turing Thesis
The above equivalence of many disparate models of computation leads to the Church-Turing Thesis, which has (at least) two different variants:

- The (equivalent) models of computation in the above theorem capture our intuitive notion of an algorithm.
- Every physically realizable computation can be simulated by one of the models in the above theorem.

This is not a precise mathematical claim, and thus cannot be formally proven, but it has stood the test of time very well, even in the face of novel technologies like quantum computers (which have yet to be built in a scalable fashion); every problem that can be solved by a quantum algorithm can also be solved by a RAM program, albeit much more slowly.

The Church-Turing Thesis in fact has a stronger variant:

Theorem 17.3. Extended Church-Turing Thesis
Every physically realizable, deterministic, and sequential model of computation can be simulated by a Word-RAM program with only a polynomial slowdown in runtime. Conversely, Word-RAM programs can be physically simulated in a deterministic and sequential manner in real time only polynomially slower than their defined runtime.

Here "deterministic" rules out both randomized and quantum computation, as both are inherently probabilistic, while "sequential" rules out parallel computation. This form of the Extended Church-Turing Thesis has stood the test of time for the approximately fifty years since it was formulated, even as computing technology has changed tremendously in that time. (Of course, the Extended Church-Turing Thesis wouldn't be true if we chose RAM model as our base model instead of Word-RAM; our choice of base model being Word-RAM, which is not too powerful, is important.)

While we do not introduce Turing Machines in CS 1200, we note that Turing Machines and Word-RAM Programs can simulate each other with only a polynomial slowdown (i.e., they are polynomially equivalent).

18 Computational Complexity

To develop a robust and clean theory for classifying problems according to computational complexity, we make two choices:

- **A problem-independent size measure:** Recall that we allowed ourselves to use different size parameters for different problems (array length n and universe size U for sorting; number n of vertices and number m of edges for graphs, number n of variable and number m of clauses for Satisfiability). To classify problems, it is convenient to simply measure the size of the input x by its length N in bits, which we call its bitlength (or sometimes just length) and denote by $|x|$. For example:
  - Array of n numbers from universe size U : $N = \Theta(n \log_2 U)$.
  - Graphs on n vertices and m edges in adjacency list notation: $N = \Theta((n+m)\log n)$.
  - 3-SAT formulas with n variables and m clauses: $N = \Theta(m \log n)$.
- **Polynomial slackness in running time:** We will only try to make coarse distinctions in running time, e.g. polynomial time vs. super-polynomial time. If the Extended Church-Turing Thesis is correct, the theory we develop will be independent of changes in computing technology; making finer distinctions like linear versus quadratic is only possible if we fix a computational model.

Definition 18.1. Complexity classes
For a function $T : \mathbb{N} \to \mathbb{R}^+$,

- $\mathsf{TIME}_{\text{search}}(T(N))$ is the class of computational problems $\Pi = (\mathcal{I}, \mathcal{O}, f)$ such that there is a Word-RAM program solving Π in time $O(T(N))$ on inputs of bit-length N.
- $\mathsf{TIME}(T(N))$ is the class of decision (i.e. yes/no) problems in $\mathsf{TIME}_{\text{search}}(T(N))$.
- (Polynomial time)

$$\mathsf{P}_{\text{search}} = \bigcup_{c \geq 0} \mathsf{TIME}_{\text{search}}\left(n^c\right), \quad \mathsf{P} = \bigcup_{c \geq 0} \mathsf{TIME}\left(n^c\right)$$

- (Exponential time)

$$\mathsf{EXP}_{\text{search}} = \bigcup_{c \geq 0} \mathsf{TIME}_{\text{search}}\left(2^{n^c}\right), \quad \mathsf{EXP} = \bigcup_{c \geq 0} \mathsf{TIME}\left(2^{n^c}\right).$$

Note that $\mathsf{P}_{\text{search}}$ and P would be the same if we replace Word-RAM with any strongly Turing-equivalent model, like Turing Machines. (Remark on terminology: what we call $\mathsf{P}_{\text{search}}$ is called Poly in the MacCormick text, and is often called FP elsewhere in the literature.)

Theorem 18.2.
We have

$$\mathsf{P}_{\text{search}} \subsetneq \mathsf{EXP}_{\text{search}}$$

and

$$\mathsf{P} \subsetneq \mathsf{EXP}.$$

19 $\mathsf{NP}_{\mathsf{search}}$ and $\mathsf{NP}_{\mathsf{search}}$-completeness

19.1 Polynomial-Time Reductions

Definition 19.1. Polynomial-Time Reductions
For computational problems Π and Γ, we write $\Pi \leq_p \Gamma$ if there is a polynomial-time reduction R from Π to Γ. That is, there is a constant $c \geq 0$ such that R runs in time at most $O\left(N^c\right)$ on inputs of length N, counting oracle calls as one time step. Equivalently, there is a constant $d \geq 0$ such that $\Pi \leq_{O\left(N^d\right), O\left(N^d\right) \times O\left(N^d\right)} \Gamma$.

Some examples of polynomial-time reduction that we've seen include:

- 3-Coloring $\leq_p$ SAT (Lecture 15)
- LongPath $\leq_p$ SAT (SRE 5)

Proof. Why does the equivalence stated at the end of the definition hold? A reduction that runs in time at most $T_R(N) = O\left(N^c\right)$ can make at most $T_R(N) = O\left(N^c\right)$ oracle calls, and each of those oracle calls is an array whose length is at most the length of its MALLOC'ed memory, which is at most $n + T_R(N) = O\left(N^c\right)$, since $n \leq N$ and assuming that $c \geq 1$ without loss of generality. The bitlength of the oracle calls is thus at most $O\left(N^c\right) \cdot w$, where w is the minimum non-crashing word length. Recall that

$$w = O\left(\log\left(\max\left\{n + T_R(N), x[0], \ldots, x[n-1]\right\}\right)\right) = O\left(\log\left(\max\left\{O\left(N^c\right), 2^N - 1\right\}\right)\right) = O(N),$$

where we used $x[i] \leq 2^N - 1$ because x, and hence each of its entries, is at most N bits long. Thus, the bitlength of the oracle queries is at most $O\left(N^c\right) \cdot O(N) = O\left(N^{c+1}\right)$, □

Lemma 19.2. $\mathsf{P}_{\mathsf{search}}$ and P are closed under polynomial-time reductions
The classes $\mathsf{P}_{\text{search}}$ and P are closed such polynomial-time reductions: Letting Π and Γ be computational problems such that $\Pi \leq_p \Gamma$, then:

1. If $\Gamma \in \mathsf{P}_{\text{search}}$, then $\Pi \in \mathsf{P}_{\text{search}}$.
2. If $\Pi \notin \mathsf{P}_{\text{search}}$, then $\mathrm{F} \notin \mathsf{P}_{\text{search}}$.

Proof.

1. Since $\Pi \leq_p \Gamma$, we have $\Pi \leq_{T_R, q \times h} \Gamma$ for $T_R(N), q(N), h(N) = O\left(N^d\right)$ for some constant d. Suppose that $\Gamma \in \mathsf{P}_{\text{search},}$, i.e. Γ can be solved in time $T_\Gamma(N) = O\left(N^b\right)$ for some $b \geq 0$. Then by Lemma 3.2 from Lecture 4 (restated in Lection 17), Π can be solved in time

$$O\left(T_R(N) + q(N) \cdot T_\Gamma(h(N))\right) = O\left(N^d + N^d \cdot \left(N^d\right)^b\right) = O\left(N^{d \cdot (b+1)}\right).$$

   So $\Pi \in \mathsf{P}_{\text{search}}$.

2. This is the contrapositive of the first item.

□

Lemma 19.3. Polynomial-time reductions compose with each other
If $\Pi \leq_p \Gamma$ and $\Gamma \leq_p \Theta$ then $\Pi \leq_p \Theta$.

These lemmas mean that we can use polynomial-time reductions both positively, to show that problems are in $\mathrm{P}_{\text{search}}$, and negatively, to give evidence that problems are not in $\mathrm{P}_{\text{search}}$. As always, the direction of the reduction is crucial!

19.2 NP

Definition 19.4. $\mathsf{NP}_{\text{search}}$
Roughly speaking, $\mathsf{NP}_{\text{search}}$ consists of the computational problems where valid outputs can be verified in polynomial time. This is a very natural requirement; what's the point in searching for something if we can't recognize when we've found it?

That is, a computational problem $\Pi = (\mathcal{I}, \mathcal{O}, f)$ is in $\mathrm{NP}_{\text{search}}$ if the following conditions hold:

- All valid outputs are of polynomial length: There is a polynomial p such that for every $x \in \mathcal{I}$ and every $y \in f(x)$, we have $|y| \leq p(|x|)$, where $|z|$ denotes the bitlength of z.
- All valid outputs are verifiable in polynomial time: There's a polynomial-time verifier V that, given $x \in \mathcal{I}$ and a potential output y, decides whether $y \in f(x)$.

Proposition 19.5. $\mathsf{NP}_{\text{search}} \subseteq \mathsf{EXP}_{\text{search}}$
As stated, we have

$$\mathsf{NP}_{\text{search}} \subseteq \mathsf{EXP}_{\text{search}}.$$

Proof. The proof is straightforward: exhanstive search! We can enumerate over all possible solutions and check if any is a valid solution. □

19.3 $\mathsf{NP}_{\text{search}}$-Completeness

Unfortunately, although it is widely conjectured, we do not know how to prove that $\mathsf{NP}_{\text{search}} \nsubseteq \mathsf{P}_{\text{search}}$. As we will see next week, this is an equivalent formulation of the famous P vs. NP problem, considered one of the most important open problems in computer science and mathematics. However, even without resolving the P vs. NP conjecture, we can give strong evidence that problems are not solvable in polynomial time by showing that they are $\mathsf{NP}_{\text{search}}$-complete.

Definition 19.6. NP-completeness, search version
A problem Γ is $\mathrm{NP}_{\text{search}}$-complete if:

1. Γ is in $\mathsf{NP}_{\text{search}}$
2. Γ is $\mathsf{NP}_{\text{search}}$-hard: For every computational problem $\Pi \in \mathsf{NP}_{\text{search}}, \Pi \leq_p \Gamma$.

We can think of the NP-complete problems as the "hardest" problems in NP.

Remarkably, if any $\mathsf{NP}_{\text{search}}$-complete problem is in $\mathsf{P}_{\text{search}}$, then all problems in $\mathsf{NP}_{\text{search}}$ are in $\mathsf{P}_{\text{search}}$.

Proposition 19.7. An $\mathsf{NP}_{\text{search}}$-complete problem is in $\mathsf{P}_{\text{search}}$ iff $\mathsf{NP}_{\text{search}} \subseteq \mathsf{P}_{\text{search}}$
Suppose Γ is $\mathsf{NP}_{\text{search}}$-complete. Then $\Gamma \in \mathsf{P}_{\text{search}}$ iff $\mathsf{NP}_{\text{search}} \subseteq \mathsf{P}_{\text{search}}$.

Proof. From the fact that Γ is $\mathsf{NP}_{\text{hard}}$ and thus amongst the hardest problems in $\mathsf{NP}_{\text{search}}$-complete, this result is intuitive. Formally, we first show that if $\Gamma \in \mathsf{P}_{\text{search}}$, then $\mathsf{NP}_{\text{search}} \subseteq \mathsf{P}_{\text{search}}$. For every problem $\Pi \in \mathsf{NP}_{\text{search}}$, we have that $\Pi \leq_p \Gamma$. The previous Lemma from Section 19.1 now ensures that $\Pi \in \mathsf{P}_{\text{search}}$. Thus, $\mathsf{NP}_{\text{search}} \subseteq \mathsf{P}_{\text{search}}$.

On the other hand, if $\mathsf{NP}_{\text{search}} \subseteq \mathsf{P}_{\text{search}}$, then $\Gamma \in \mathsf{P}_{\text{search}}$, using the fact that $\Gamma \in \mathsf{NP}_{\text{search}}$. This completes the proof. □

Further, there are natural $\mathsf{NP}_{\text{search}}$-complete problems, including CNF-Satisfiability.

Theorem 19.8. Cook-Levin Theorem
SAT is $\mathsf{NP}_{\text{search}}$-complete.

Proof. We will return to a proof of the Cook-Levin Theorem later in the course. As a remark on interpretation: this result can be interpreted as strong evidence that SAT is not solvable in polynomial time; if it were, then the previous proposition would imply that every problem in $\mathsf{NP}_{\text{search}}$ would be solvable in polynomial time. □

20 $\mathsf{NP}_{\text{search}}$-Completeness via Mapping Reductions

20.1 3-SAT

Once we have one $\mathrm{NP}_{\text{search}}$-complete problem, we can get others via reductions from it.

Definition 20.1. 3-SAT problem
The computational problem 3-SAT is obtained when we restrict the number of literals in each clause of SAT.

- **Input:** A CNF formula φ on n variables $z_0, \ldots z_{n-1}$ in which each clause has width at most 3 (i.e. contains at most 3 literals)
- **Output:** An $\alpha \in \{0,1\}^n$ such that $\varphi(\alpha) = 1$ (if one exists)

Theorem 20.2. 3-SAT is $\mathsf{NP}_{\text{search}}$-complete.
As stated, 3-SAT is $\mathsf{NP}_{\text{search}}$-complete.

20.2 Mapping Reductions

The usual strategy for proving that a problem Γ in $\mathsf{NP}_{\text{search}}$ is also $\mathsf{NP}_{\text{search}}$-hard (and hence $\mathsf{NP}_{\text{search}}$-complete) is called a mapping reduction.

1. Pick a known $\mathsf{NP}_{\text{search}}$-complete problem Π to try to reduce to Γ. Typically, we might try to pick a problem Π that seems as similar as possible to Γ, or which has been used to prove that problems similar to Γ are $\mathsf{NP}_{\text{search}}$-complete. Otherwise 3-SAT is often a good fallback option.
2. Come up with an algorithm R mapping instances x of Π to instances $R(x)$ of Γ. If Π is 3 SAT, this will often involve designing "variable gadgets" that force solutions to $R(x)$ to encode true/false assignments to variables of x and "clause gadgets" that force these assignments to satisfy each of the clauses of x. We will see an example in the next lecture.
3. Show that R runs in polynomial time.
4. Show that if x has a valid answer, then so does $R(x)$. That is, we can transform valid answers to x to valid answers to $R(x)$.
5. Conversely, show that if $R(x)$ has an answer, then so does x. Moreover, we can transform valid answers to $R(x)$ into valid answers to x through a polynomial time algorithm S. This transformation needs to be efficient (in contrast to Item 4 because it has to be carried out by our reduction).

A formal definition of mapping reductions follows (but we won't expect you to use this formalism, you can stick with the general definition of polynomial-time reductions):

Definition 20.3. Polynomial-time mapping reduction
Let $\Pi = (\mathcal{I}, \mathcal{O}, f)$ and $\Gamma = (\mathcal{J}, \mathcal{P}, g)$ be search problems. A polynomial-time mapping reduction from Π to Γ consists of two polynomial-time algorithms R and S such that for every $x \in \mathcal{I}$:

1. $R(x) \in \mathcal{J}$.
2. If $f(x) \neq \emptyset$, then $g(R(x)) \neq \emptyset$. That is, if an input x has some correct answer in $\mathcal{O}$, then the transformed $R(x)$ has corresponding a correct answer in $\mathcal{P}$.
3. $\forall y \in g(R(x))$, we have $S(x, y) \in f(x)$.

Note that the above outline only proves $\mathsf{NP}_{\text{search}}$-hardness; a proof that Γ is $\mathsf{NP}_{\text{search}}$-complete should also check that it's in $\mathsf{NP}_{\text{search}}$.

20.3 Independent Set

Theorem 20.4. IndependentSet is $\text{NP}_{\text{search}}$-complete
As stated, IndependentSet is $\text{NP}_{\text{search}}$-complete.

Proof. A natural approach is a mapping reduction from 3-SAT. □

20.4 Longest Path

Theorem 20.5. LongPath and HamiltonianPath are $\mathsf{NP}_{\text{search}}$-complete
The LongPath problem is as follows:

- **Input**: A digraph $G = (V, E)$, two vertices $s, t \in V$, and a path-length $k \in \mathbb{N}$
- **Output**: A path from s to t in G of length k, if one exists

It may be shown that LongPath is $\mathsf{NP}_{\text{search}}$-complete, even in the special case where $k = n - 1$, i.e. the path needs to visit all vertices in the graph.

21 The P vs. NP Problem

In this lecture, we translate our discussion about search problems into discussion about decision problems, which are commonly discussed.

Definition 21.1. Decision problem
A computational problem $\Pi = (\mathcal{I}, \mathcal{O}, f)$ is a decision problem if $\mathcal{O} = \{\text{yes}, \text{no}\}$ and for every $x \in \mathcal{I}, |f(x)| = 1$.

By definition,

$$\mathsf{P} = \{\Pi : \Pi \in \mathsf{P}_{\text{search}} \text{ and } \Pi \text{ is a decision problem}\}$$
$$\mathsf{EXP} = \{\Pi : \Pi \in \text{EXP}_{\text{search}} \text{ and } \Pi \text{ is a decision problem}\}.$$

Definition 21.2. NP
A decision problem $\Pi = (\mathcal{I}, \{\text{yes, no}\}, f)$ is in NP if there is a computational problem $\Gamma = (\mathcal{I}, \mathcal{O}, g) \in \mathsf{NP}_{\text{search}}$ such that for all $x \in \mathcal{I}$, we have:

$$f(x) = \{\text{yes}\} \iff g(x) \neq \emptyset$$
$$f(x) = \{\text{no}\} \iff g(x) = \emptyset$$

Intuitively, NP consists of decision problems Π where a yes answer has a short, efficiently verifiable proof. Indeed, we can prove that $f(x) = \{\text{yes}\}$ by giving a solution $y \in g(x)$, which is of at most polynomial length and is verifiable in polynomial time.

Lemma 21.3. $\mathsf{P} \subseteq \mathsf{NP}$
We have

$$\mathsf{P} \subseteq \mathsf{NP}.$$

Proof. This is an intuitive result, as to verify a yes or no on some input, we can simply use a polynomial-time algorithm to solve the decision problem, and compare the output (also a yes or no) to verify the answer.

Formally, Let $\Pi = (\mathcal{I}, \{\text{yes}, \text{no}\}, f)$ be an arbitrary computational problem in P. Our goal is to come up with a computational problem $\Gamma = (\mathcal{I}, \mathcal{O}, g)$ in $\mathsf{NP}_{\text{search}}$ that satisfies the requirements of our definition of NP. We do this by setting $g(x) = f(x) \cap \{\text{yes}\}$ and $\mathcal{O} = \{\text{yes}\}$.

Thus, $f(x) = \{\text{yes}\}$ iff $g(x) \neq \emptyset$, and it can be verified that $\Gamma \in \mathsf{NP}_{\text{search}}$. (The verifier $V(x, y)$ for Γ can check that $y = \text{yes}$ and that the polynomial-time algorithm for Π outputs yes on x.) Thus, we conclude that $\Pi \in \mathsf{NP}$. □

Theorem 21.4. Search vs. Decision
$\mathsf{NP} = \mathsf{P}$ if and only if $\mathsf{NP}_{\text{search}} \subseteq \mathsf{P}_{\text{search}}$.

Lemma 21.5. SAT $\leq_p$ SAT-Decision
We have

$$\text{SAT} \leq_p \text{SAT-Decision}.$$

Definition 21.6. NP-complete
As expected, we say a decision problem Π is NP-complete if $\Pi \in \mathsf{NP}$, and Π is NP-hard, meaning $\Gamma \leq_p \Pi$ for every problem $\Gamma \in \mathsf{NP}$.

All of the $\mathsf{NP}_{\text{search}}$-completeness proofs we have seen are also NP-completeness proofs of the corresponding decision problems:

Theorem 21.7. List of NP-complete problems
SAT-Decision, 3-SAT-Decision, IndependentSet-Decision, SubsetSum-Decision, LongPathDecision, and 3D-Matching-Decision are NP-complete.

22 Uncomputability by Reductions

22.1 Universal Programs

In this section, we study algorithms for analyzing programs, i.e., computational problems whose inputs are not arrays of integers, or graphs, or logical formulas, but ones whose inputs are themselves programs.

Theorem 22.1. Universal RAM simulator in RAM
There is a RAM program U such that for every RAM program P and input x, U halts on input (P, x) iff P halts on x and if so, $U(P, x) = P(x)$. Moreover, $T_U((P, x)) = O\left(T_P(x) + |P| + |x|\right)$.

22.2 The Halting Problem

Definition 22.2. Halting Problem
We define the halting problem for RAM Programs by:

- **Input:** A RAM program P and an input x
- **Output:** yes (i.e. accept or 1) if P halts on input x, and no otherwise

Analogously, we define the Word-RAM program version of the problem by:

- **Input:** A Word-RAM program P and an input x
- **Output:** yes (i.e. accept or 1) if there is a word length w such that $P[w]$ halts on input x, and no otherwise

Theorem 22.3. Unsolvability of halting
There is no algorithm (on any model of computation, according to the Church-Turing thesis) that solves the Halting Problem for RAM Programs or for Word-RAM programs.

22.3 Unsolvability

In previous lectures, we were concerned with the complexity classes of problems. Now, we are only concerned with solvability, regardless of the necessary computational runtime.

Definition 22.4. Solvability
Let $\Pi = (\mathcal{I}, \mathcal{O}, f)$ be a computational problem. We say that Π is solvable if there exists RAM program P that solves Π. Otherwise we say that Π is unsolvable.

In the cases that the problem Π amounts to computing a function (i.e., $|f(x)| = 1$ for all x), we say the function is computable or uncomputable. Restricting further to decision problems, we say the problem is decidable or undecidable.

Our study of reductions relies on our lemma about reductions:

Lemma 22.5. Using reductions to demonstrate unsolvability
Let Π and Γ be computational problems such that $\Pi \leq \Gamma$. Then if Π is unsolvable, then Γ is unsolvable.

22.4 Other unsolvable problems via reduction

Definition 22.6. HaltsOnEmpty problem (RAM version)
We modify the halting problem by considering, in particular, the empty input ε:

- **Input:** A RAM program Q
- **Output:** yes (i.e. accept or 1) if Q halts on the empty input ε, and no otherwise

Theorem 22.7. HaltsOnEmpty-RAM is unsolvable
HaltsOnEmpty-RAM is unsolvable.

Proof. We reduce the Halting Problem-RAM to HaltsOnEmpty-RAM to □

23 Unsolvability of the Halting Problem

Omitted.

24 Satisfiability Modulo Theories and Cook–Levin

Omitted.