CS120: Intro. to Algorithms and their Limitations Hesterberg & Vadhan

Section 4: Midterm Review

Harvard SEAS - Fall 2022 *Sept. 29, 2022*

Contents

Conceptual Review

1 Mathematical Tools

1.1 General Proof and Disproof Types

A proof is a sequence of statements, each of which is a logical consequence of the previous ones, starting from definitions and obviously-true statements ("axioms") and ending with what you're trying to prove. Some mathematical techniques that are often helpful are:

- Casework
- Weak induction
- Strong induction
- Contradiction
- Counterexample

Note that these types are not necessarily mutually exclusive, and some proofs don't use any of them (sometimes called "direct proofs").

Tip: When disproving a statement with a counterexample, make your example as specific and detailed as possible! This ensures that a) there is a concrete instance of the situation you have described, and b) you don't accidentally miss a crucial component of your counterexample.

1.2 Induction

Let $P(n)$ be the predicate, that is, the statement which we want to prove true for all $n \geq a$ for some integer a. If:

1. **Base Case:** $P(a)$ is true.
2. **Inductive Hypothesis:** Assume that the statement $P(k)$ for $k = n - 1$.
3. **Inductive Step:** Prove that the statement $P(k)$ is also true for $k = n$.

Then, by the principle of mathematical induction, $P(n)$ holds true for all $n \geq a$.

Tip: For best practices, be sure to explicitly label base case(s) and inductive steps.

1.3 Strong Induction

Strong induction is a variant of induction, which uses a stronger assumption for the inductive step (ii). Rather than assuming that $P(k)$ is true for just $k = n - 1$, we must assume that $P(k)$ is true for all $k < n$.

In application of recurrences that rely on two or more preceding cases, strong induction can be useful for proofs.

Concept Check: Can any proof done with strong induction be done with weak induction? What about the converse?

1.4 Probability

- Random variable. Ex: die/dice, or a coin toss
- Expected value: Weighted average of all outcomes. Intuitively, this is equivalent to the mean of a large of number of independently chosen outcomes of a single random variable. E.g.: The expected value when I roll a die is 3.5, because all six faces are equally probable to be rolled, and the mean of the set {1,2,3,4,5,6} is 3.5.
- Linearity of expectation: $\mathbb{E}[X+Y] = \mathbb{E}[X] + \mathbb{E}[Y]$, where X and Y are random variables, even if they are dependent!

Question 1.1. 1. What is the expected value of a variable that's 1 with probability p and 0 otherwise (called a "Bernoulli random variable with parameter p")?

2. We throw two 6-sided dice independently. Let X correspond to the sum of the numbers rolled on the dice, and let Y correspond to their product. What is $\mathbb{E}[X+Y]$?

1.5 Recursion

A recurrence is a function defined in terms of other values of that function. A simple example is the Fibonacci sequence:

$$\text{Fib}(n+2) = \text{Fib}(n+1) + \text{Fib}(n);\ \text{Base cases: Fib}(0) = 1, \text{Fib}(1) = 1$$

In many cases, it is more useful for us to derive a closed-form expression of the recurrence. A common way to find the value of a recurrence in closed form is to "unroll" it. Consider this example[1]:

$$T(n) = 2T(n-1) + 5;\ T(1) = 10$$

We begin unrolling by applying our recurrence over and over again:

$$\begin{aligned} T(n) &= 2 \cdot T(n-1) + 5 \\ &= 2(2 \cdot T(n-2) + 5) + 5 = 2^2 \cdot T(n-2) + 2 \cdot 5 + 5 \\ &= 2^2(2 \cdot T(n-3) + 5) + 2 \cdot 5 + 5 = 2^3 \cdot T(n-3) + 2^2 \cdot 5 + 2 \cdot 5 + 5 \\ &\dots \end{aligned}$$

We look for a pattern:

$$T(n) = 2^i T(n-i) + \sum_{k=0}^{i-1} 2^k \cdot 5$$

[1] Fleck, 2009

When will this expression look like our original case of $T(1) = 10$? This will look like $T(1)$ when $i = n - 1$:

$$T(n) = 2^{n-1}T(1) + \sum_{k=0}^{n-2} 2^k \cdot 5$$
$$T(n) = 10 \cdot 2^{n-1} + \sum_{k=0}^{n-2} 2^k \cdot 5$$
$$T(n) = 10 \cdot 2^{n-1} + 5 \cdot (2^{n-1} - 1)$$
$$T(n) = 15 \cdot 2^{n-1} - 5$$

And now we have a closed-form expression.

2 Asymptotic Analysis

Remember that, when discussing the running time, $T(n)$, of an algorithm in this class, we describe the *worst-case* running time of the algorithm on an input of size at most n. To help us describe these running time functions, we use several types of notation:

Definition 2.1. Let $f, g : \mathbb{N} \to \mathbb{R}^+$. We say:

- $f = O(g)$ if there is a constant $c > 0$ such that $f(n) \leq c \cdot g(n)$ for all sufficiently large n.
- $f = \Omega(g)$ if there is a constant $c > 0$ such that $f(n) \geq c \cdot g(n)$ for all sufficiently large n. Equivalently, $g = O(f)$.
- $f = \Theta(g)$ if $f = O(g)$ and $f = \Omega(g)$.
- $f = o(g)$ if for every constant $c > 0$, we have $f(n) < c \cdot g(n)$ for all sufficiently large n. Equivalently, $\lim_{n\to\infty} f(n)/g(n) = 0$.
- $f = \omega(g)$ if for every constant $c > 0$, we have $f(n) > c \cdot g(n)$ for all sufficiently large n. Equivalently, $g = o(f)$. Equivalently, $\lim_{n\to\infty} f(n)/g(n) = \infty$.

Concept Check: Does little-O imply big-O? What about the converse?

Concept Check: Does big-O and NOT little-O imply Θ? In other words, if $f = O(g)$, and $f \neq o(g)$, is it true that $f = \Theta(g)$?

Question 2.2. True/False. For all questions, functions are from $\mathbb{N}$ to $\mathbb{N}$.

- Suppose $f(n) = o(g(n))$ and $h(n) = O(g(n))$. Then $f(n)h(n) = o(g^2(n))$.
- Suppose
$$f(n) = \begin{cases} n & n \text{ even} \\ n^2 & n \text{ odd.} \end{cases}$$
Then there is no function g where $f(n) = \omega(g(n))$.

3 Computational Problems

Definition 3.1. A *computational problem* is a triple $\Pi = (\mathcal{I}, \mathcal{O}, f)$ where

- $\mathcal{I}$ is the set of inputs/instances (typically infinity).
- $\mathcal{O}$ is the set of outputs.
- $f(x)$ is the set of correct answers for input x.
- For each $x \in \mathcal{I}$, $f(x) \subseteq \mathcal{O}$.

Definition 3.2. Let Π be a computational problem and let A be an algorithm. We say that A *solves* Π if

- For every $x \in \mathcal{I}$ such that $f(x) \neq \emptyset$, $A(x) \in f(x)$.
- $\exists$ a special symbol $\perp \notin \mathcal{O}$ such that for all $x \in \mathcal{I}$ such that $f(x) = \emptyset$, we have $A(x) = \perp$.

Examples of computational problems we have seen in class include the sorting problem, interval-scheduling problem, and duplicate searching, etc.
Tip: Be careful not to mix up $\mathcal{O}$, $f(x)$, and $A(x)$! $\mathcal{O}$ represents all *possible* outputs from the computational problem. $f(x)$ gives us the **set** of valid answers from an input x, and $f(x) \subseteq \mathcal{O}$. An algorithm A that correctly solves the problem must return $A(x)$ where, if $A(x) \neq \perp$, $A(x) \in f(x)$.

Concept Check 1: Which, among $f(x), A(x),$ and $\mathcal{O}$, are necessarily sets?

Concept Check 2: Consider the sorting computational problem. What is $f(x)$ when $x = ((0, A), (1, B), (3, C), (1, D), (4, E))$? (Note that these are key-item pairs.) If $A(x) = ((0, A), (1, B), (1, D), (3, C), (4, E))$, has A solved the problem for x?

4 Sorting Algorithms

Input : An array A of key-value pairs $((K_0, V_0), \ldots, (K_{n-1}, V_{n-1}))$, where each key $K_i \in \mathbb{R}$

Output : An array A' of key-value pairs $((K'_0, V'_0), \ldots, (K'_{n-1}, V'_{n-1}))$ that is a *valid sorting* of A. That is, A' should be:

1. sorted by key values, i.e. $K'_0 \leq K'_1 \leq \cdots K'_{n-1}$. and
2. a permutation of A, i.e. $\exists$ a permutation $\pi : [n] \to [n]$ such that $(K'_i, V'_i) = (K_{\pi(i)}, V_{\pi(i)})$ for $i = 0, \ldots, n-1$.

Computational Problem Sorting

Concept Check: Must the keys K_i be real numbers? What about the values V_i? Hence, if you are creating an array of key-value pairs, where should you place the numbers that you want to sort?

Examples of sorting algorithms we've seen in class:

- Exhaustive search sort: $O(n! \cdot n)$
- Insertion sort: $O(n^2)$
- Merge sort: $O(n \log n)$
- Counting sort: $O(U + n)$
- Radix sort: $O(n + n(\log U)/(\log n))$

One particular class of sorting algorithms we discussed is **comparison-based sorting**, where the only operations we do to the keys are comparing and copying them (not, say, looking at bits of them).

Theorem 4.1. *If A is a comparison-based sorting algorithm that makes at most $T(n)$ comparisons on every array of size n, then $T(n) = \Omega(n \log n)$.*

Concept Check: Which of the algorithms above are comparison-based? Which are stable sorting algorithms?

Question 4.2. It's the day of Harvard-Yale, and you have a side bet: the big Tug-o-War match will result in a perfect tie. To ensure you win your bet, you want to fix the teams (of two players each) such that the sum of Harvard's players' weights is exactly equal to the sum of Yale's players' weights. You are given the lists $H = (h_0, \ldots, h_{n-1})$ and $Y = (y_0, \ldots, y_{n-1})$, of weights of all players on both teams' rosters, and tasked to determine if there are 2 players from each team such that $h_a + h_b = y_c + y_d$. Show that we can solve this problem in time $O(n^2 \log n)$. Hint: how much time does it take to sort two arrays of size $O(n^2)$?

5 Static Data Structures

Definition 5.1. 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 a set of *queries*, and
- for every $x \in \mathcal{I}$ and $q \in \mathcal{Q}$, $f(x, q) \subseteq \mathcal{O}$ is a set of *valid answers*.

Definition 5.2. For a (static) data structure problem $\Pi = (\mathcal{I}, \mathcal{O}, \mathcal{Q}, f)$, an *implementation* 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$(x), q) = \perp$.

In Lecture 4, we saw an example of how solving the static predecessors and successors problem allowed us to also solve the interval-scheduling problem.

6 Dynamic Data Structures

Definition 6.1. 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*.

Concept Check: How are dynamic data structure problems different?

Definition 6.2. For a dynamic data structure problem $\Pi = (\mathcal{I}, \mathcal{O}, \mathcal{Q}, \mathcal{U}, f)$, an *implementation* is a triple of algorithms $(\text{Preprocess}, \text{EvalQ}, \text{EvalU})$ 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 $f(x, u_0, u_1, \ldots, u_{m-1}) \neq \emptyset$, we have:

$$\text{EvalQ}(\text{EvalU}(\cdots \text{EvalU}(\text{EvalU}(\text{Preprocess}(x), u_0), u_1), \ldots, u_{m-1}), q) \in f(x, u_0, u_1, \ldots, u_{m-1}, q),$$

else

$$\text{EvalQ}(\text{EvalU}(\cdots \text{EvalU}(\text{EvalU}(\text{Preprocess}(x), u_0), u_1), \ldots, u_{m-1}), q) = \emptyset.$$

6.1 Binary Search Trees (BSTs)

Definition 6.3. A *binary search tree (BST)* is a recursive data structure. Every nonempty BST has a root vertex r, and every vertex v has:

- a key K_v
- a value V_v
- a pointer to a left child v_L, which is another binary tree (possibly `None`, the empty BST)
- a pointer to a right child v_R, which is another binary tree (possibly `None`, the empty BST)

Of course, we require that the parents of v_L and v_R (if they exist) are v. 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.

Theorem 6.4. *Given a binary search tree of height (equivalently, depth) h, all of the following operations (queries and updates) can be performed in time $O(h)$:*

1. *Search queries: given K, return a matching pair (K, V) in the dataset specified by the updates (if one exists)*

2. *Insertion updates: given a key-value pair (K, V), add (K, V) to the dataset.*

3. *Minimum queries: return a pair (K, V) with the smallest value of K in the dataset specified by the updates.*

4. *Maximum queries: return a pair (K, V) with the largest value of K in the dataset specified by the updates*

5. *Next-smaller queries: given a key K', return a key-value pair (K, V) in the dataset with maximum key such that $K < K'$, or $\perp$ if no such key K exists*

6. *Next-larger queries: analogous to next-smaller queries*

7. *Deletion updates: given a key K, delete a matching pair (K, V) from the dataset specified by the updates (if one exists)*

Concept Check: A certain BST contains a key-value pair $(1, 210)$, and no key is less than 1. A Minimum query returns $(1, 1)$. Is this necessarily wrong?

6.2 Height-balanced BSTs (AVLs)

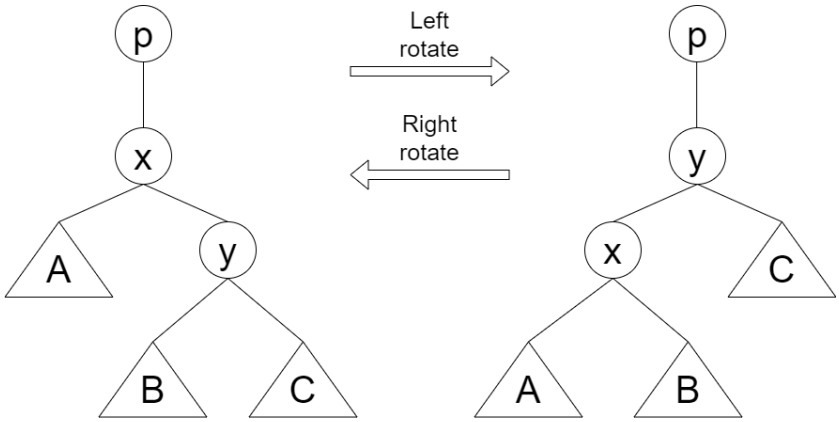
Definition 6.5 (AVL Trees)**.** An *AVL Tree* is a binary search tree in which:

- Every node has an additional attribute containing its *height* in the tree (length of the longest path from the node to a leaf). ("data structure augmentation")
- Every pair of siblings in the tree have heights differing by at most 1 (where we define the height of the an empty subtree to be -1). ("height-balanced") f

Lemma 6.6. *Every AVL Tree with n nodes has height (=depth) at most $2 \log_2 n$.*

Concept Check: Do AVL trees satisfy the BST property?

We discussed rotations on an AVL tree:



Theorem 6.7. *We can insert a new key-value pair into an AVL Tree while preserving the AVL Tree property in time $O(\log n)$, using $O(\log n)$ rotations.*

Concept Check: How much time does a rotation take?

Concept Check: What is the difference between the size and height attribute?

Question 6.8. Show that among all binary trees of depth at most d, the complete binary tree of depth d has the largest number of leaves.

7 Reductions

Definition 7.1. Let $\Pi = (\mathcal{I}, \mathcal{O}, f)$ and $\Gamma = (\mathcal{J}, \mathcal{Q}, g)$ be two computational problems. A *reduction* from Π to Γ is an algorithm that solves Π using as a subroutine a(ny) *oracle* that solves Γ. An oracle is a function that, given any input $x \in \mathcal{I}$ to a computational problem $P = (\mathcal{I}, \mathcal{O}, f)$, returns an element of $f(x)$.

If there exists a reduction from Π to Γ, then we write $\Pi \leq \Gamma$ (read "Pi reduces to Gamma").

Lemma 7.2. *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 there does not exist an algorithm solving Γ.*
3. *If there exists an algorithm solving Γ with runtime $g(n)$, and $\Pi \leq_{T,f} \Gamma$, then there exists an algorithm solving Π with runtime $O(T(n) + g(f(n)))$.*
4. *If there does not exists an algorithm solving Π with runtime $O(T(n)+g(f(n))$, and $\Pi \leq_{T,f} \Gamma$, then there does not exist an algorithm solving Γ with runtime $O(g(n))$.*

In Lecture 3, we saw a key example of a reduction: DuplicateSearch $\leq_{O(n),n}$ Sorting.

Concept Check: If you know that $\Pi \leq_{T,f} \Gamma$, can you deduce something about the runtime of the oracle?

Concept Check: If $\Pi \leq \Gamma$, then which problem can be solved using an oracle for the other problem as a subroutine?

Question 7.3. Suppose we are given an array of the lower left and upper right corner of n rectangles as input:

$$((p_0, q_0), \ldots, (p_{n-1}, q_{n-1})).$$

(Note that each p_i and each q_i is a pair of natural numbers). Show that we can determine if any two rectangles have the same area in time $O(n \log n)$.

When designing a reduction, make sure to think of it as a 3-step process:

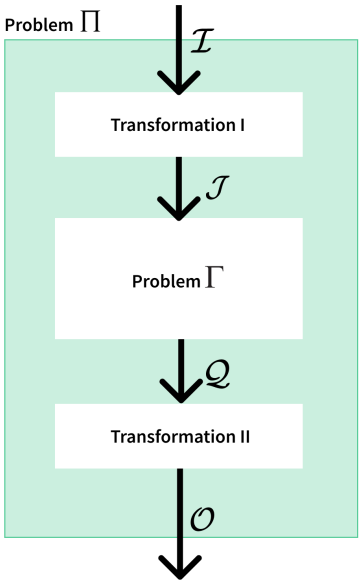


Figure 1: A reduction from Π to Γ which calls the oracle for Γ exactly once.

8 RAM Model

Definition 8.1 (RAM Programs)**.** 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*), chosen from the following:

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

Concept Check: What is the running time of a RAM program?

Concept Check: Can a RAM program contain the line "IF `var` == 3, GOTO k", where $k \in \{0, 1, \ldots, \ell\}$.

Question 8.2. Suppose we augment the RAM model with the JLE ("jump on less than or equals") operation "IF $\texttt{var}_0 \leq \texttt{var}_1$ GOTO k."

Show that for every JLE-augmented RAM program P, there is an ordinary RAM program P' such that for every input x, $P'(x) = P(x)$ and $\text{Time}_{P'}(x) = O(\text{Time}_P(x))$.

Given your answer, are JLE-augmented RAM programs computationally more powerful than RAM programs?

Concept Check: Are JLE-augmented RAM programs (from Question 8.2) computationally more powerful than RAM programs?

8.1 Word RAM Model

Definition 8.3. The *Word RAM Model* is defined like the RAM Model except that it has a dynamic *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.
- Operations: Addition and multiplication are redefined from RAM Model to return $2^w - 1$ if the result would be $\geq 2^w$.
- Initial settings: When a computation is started on an input x, which is an array consisting of n natural numbers, the memory size is taken to be $S = n$, and word length is taken to be $w = \lfloor \log \max\{S, x[0], \ldots, x[n-1]\} \rfloor + 1$. (This setting is to ensure that $S, x[0], \ldots, x[n-1]$ are all strictly smaller than 2^w and hence fit in one word.)
- Increasing S and w: If the algorithm needs to increase its memory size beyond S, it can issue a MALLOC command, which increments S by 1, sets $M[S-1] = 0$, and if $S = 2^w$, it also increments w.

The current values of the word length and memory size are also made available to the algorithm in read-only variables `word_len` and `mem_size`.

Theorem 8.4. *1. For every RAM program P, there is a Word-RAM Program P' such that P' halts on x iff P halts on x, and if they halt, then $P'(x) = P(x)$ and*

$$\text{Time}_{P'}(x) = O\left((\text{Time}_P(x) + n + S) \cdot \left(\frac{\log M}{w_0}\right)^{O(1)}\right),$$

where n is the length of the input x, S is the largest memory location accessed by P on input x, M is the largest number computed by P on input x, and $w_0 = \lfloor \log \max\{n, x[0], \ldots, x[n-1]\} \rfloor + 1$.

2. *For every Word-RAM program P, there is a RAM program P' such that P' halts on x iff P halts on x, and if they halt, then $P'(x) = P(x)$ and*

$$\text{Time}_{P'}(x) = O\left(\text{Time}_P(x) + n + w_0\right),$$

where n is the length of x and w_0 is the initial word size of P on input x.

Concept Check: Is RAM as computationally powerful as Word RAM? What about the converse? Is Python more computationally powerful than either of them?

9 Randomized Algorithms

There are two different flavors of randomized algorithms:

- *Las Vegas Algorithms:* Always output a correct answer, but their running time depends on their random choices. In the worst case, Las Vegas algorithms can have infinite running time. Typically, we try to bound their *expected* running time. That is, we say the *(worst-case) expected running time* of A is

$$T(n) = \max_{x:\text{size}(x)\leq n} \text{E}[\text{Time}_A(x)],$$

  where $\text{Time}_A(x)$ is the random variable denoting the runtime of A on x.

- *Monte Carlo Algorithms:* 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 $\Pi = (\mathcal{I}, f)$ *with error probability* p if

$$\text{for every } x \in \mathcal{I},\ \Pr[A(x) \in f(x)] \geq 1 - p$$

  Think of the error probability as a small constant, like $p = .01$. Typically this constant can be reduced to an astronomically small value by running the algorithm several times independently.

Prevailing conjecture (based on the theory of pseudorandom number generators) is: **randomness is not necessary for polynomial-time computation.** More precisely, we believe a $T(n)$ time Monte-Carlo algorithm *implies* a deterministic algorithm in time $O(n \cdot T(n))$, so we can convert any randomized algorithm into a deterministic one. (You do not need to know details about this conjecture beyond its statement for the purposes of this class.)

Concept Check: Which is preferable – Monte Carlo or Las Vegas?

9.1 Quick Select

Input	: An array A of key-value pairs $((K_0, V_0), \ldots, (K_{n-1}, V_{n-1}))$, where each key $K_j \in \mathbb{N}$, and a *rank* $i \in [n]$
Output	: A key-value pair (K_j, V_j) such that K_j is an i'th smallest key. That is, there are at most i values of k such that $K_k < K_j$ and there are at most $n - i - 1$ values of k such that $K_k > K_j$.

Computational Problem Selection

In particular, when $i = (n-1)/2$, we need to find the *median* key in the dataset.

Theorem 9.1. *There is a randomized algorithm QuickSelect that always solves Selection correctly, and has (worst-case) expected running time $O(n)$.*

Question 9.2. A student in the class dislikes randomized algorithms, so they create NonRandomizedQuickSelect. In this algorithm, we always choose the pivot to be the *first* element in the input array (and there are no other changes to the algorithm). Determine what the worst-case runtime of this algorithm is.

10 Randomized Data Structures

A Monte Carlo data structure:

- Preprocess(U, m): Initialize an array A of size m. In addition, choose a random hash function $h : [U] \to [m]$ from the universe $[U]$ to $[m]$.
- Insert(K, V): Place (K, V) into the linked list at slot $A[h(K)]$.
- Delete(K): Remove an element from the linked list at slot $A[h(K)]$.
- Search(K): Return the head of the linked list at $A[h(K)]$.

If there are two distinct keys K_1, K_2 with $h(K_1) = h(K_2)$, on the query Search(K_1) we could return an key-value pair (K_2, V). To bound this error probability, consider a query Search(K). In order for us to return the wrong value, we must have inserted a distinct key that hashes to the same value as $h(K)$. If we've inserted items with keys $K_0, \ldots, K_{n-1}$ that are different than K, we have:

$$\begin{aligned}\Pr[\text{Search}(K) \text{ returns incorrect value}] &\leq \Pr[h(K) \in \{h(K_0), \ldots, h(K_{n-1})\}] \\ &\leq \sum_{i=0}^{n-1} \Pr[h(K) = h(K_i)] \\ &= n \cdot \frac{1}{m}\end{aligned}$$

where in the final line we used that h was a random mapping from $[U]$ to $[m]$.

A Las Vegas data structure ("Hash Table"): The same data structure as above, except the linked list at every array index stores (K, V) pairs. Then when we query Search(K), go to the linked list $A[h(K)]$ and return the first element of the list that has the correct key K (and if none do, return $\perp$).

Here, we bound the *expected runtime* via the same analysis as before, because if no elements collide (the event we bound above) the additional linked list checking will only be an $O(1)$ slowdown. Quantitatively, we can show an expected runtime of $O(1 + n/m)$. We call $\alpha = n/m$ the *load* of the table, so the runtime is $O(1 + \alpha)$. Notice that here we get $O(1)$ expected time even if $n > m$, provided that $m = \Omega(n)$.

Question 10.1. The World Surf League (WSL) wishes to create and maintain a database of all professional surfers in the year 2030, when surfing has become so popular that they need to use effective algorithms to manage the huge amount of data. The data for each surfer is $(\texttt{name}, \texttt{points}, \texttt{bio})$, where `name` is the (unique) name of each surfer, `points` is the number of points they have accumulated (you may also assume there are no ties), and `bio` is a biography of the surfer.

For each of the scenarios below, select the best algorithm or data structure for the WSL to use from among: (a) computing and storing a sorted array, (b) store in a binary search tree (balanced and possibly augmented), (c) store in a hash table, (d) randomized quickselect. Briefly justify your answers.

1. Given a surfer's `name`, the WSL TV commentators want to be able to instantly pull up that surfer's `points` and `bio`. The `points` and `bio` may also be often updated throughout the season. (`points` are accumulated by winning heats in surf competitions, with different heats being worth different numbers of points.)

2. Halfway through the season, the WSL decides to cut the bottom half of the surfers (according to `points`) out of remaining competitions.

3. At the end of the season, the WSL needs to publish a ranking of all of the surfers according to their `points`, to determine the year's world champions, who qualifies to be on the Championship tour in the following year, etc.

4. To create more excitement, the WSL decides that when surfers who have adjacent rankings are competing against each other in a heat, the number of points for the heat will be doubled. To implement this, they need to quickly determine whether two surfers are adjacent to each other in the rankings.