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

Section 1: Asymptotic Analysis, Sorting, Computational Problems

Harvard SEAS - Fall 2023 *Sept. 14, 2023*

1 Asymptotic Analysis

Asymptotic notation is often used in computer science to analyze the time complexity of algorithms. Asymptotic notation analyzes how the runtime of the algorithm changes as the input size becomes arbitrarily large. Recall the following definitions from class:

Definition 1.1. 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 $f(n) \leq c \cdot g(n)$ 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 $f(n) \geq c \cdot g(n)$ for all sufficiently large n. Equivalently, $g = o(f)$.

1.1 Concept Check

Determine whether or not the following statements are True or False.

1. $f = o(g)$ implies $f = O(g)$. What about the converse?

2. $f = \Theta(g)$ implies $f = O(g)$ and $f = \Omega(g)$. What about the converse?

3. $f = o(g)$ implies $g = \Omega(f)$.

4. $f = O(g)$ implies $g = \omega(f)$.

1.2 Problems

Question 1.2. For each of the following two claims, either justify why the statement holds (for all f, g) or provide a counterexample. In all cases, take the domain of the functions f and g to be the natural numbers (rather than the positive reals), and assume $f(n), g(n) \geq 1$ for all sufficiently large n (so that the logarithms are nonnegative).

- If $\log(f(n)) = O(\log(g(n)))$, then $f(n) = O(g(n))$.

- If $g(n) = o(f(n))$, then $f(n) + g(n) = \Theta(f(n))$.

2 Counting Sort Review

We present Counting Sort here for key-value pairs, but it may also be used for just sorting an array of keys from the universe U.

Input : An array $A = ((K_0, V_0), \ldots, (K_{n-1}, V_{n-1}))$, where each $K_i \in [U]$
Output : A valid sorting of A
1 Initialize an array C of length U, such that each entry of C is the start of an empty linked list.
2 **foreach** $i = 0, \ldots, n-1$ **do**
3 | Append (K_i, V_i) to the linked list $C[K_i]$.
4 Form an array A that contains the elements of $C[0]$, followed by the elements of $C[1]$, followed by the elements of $C[2]$, ...
5 **return** A

Algorithm 1: Counting Sort with Values

To show the correctness of Algorithm 1, we observe that after the loop, for each $j \in [U]$, the linked list $C[j]$ contains exactly the key-value pairs whose key equals j. Thus concatenating these linked lists into a single array will be a valid sorting of the input array.

Question 2.1. Consider the Universe $U = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10\}$. Perform Counting Sort on the array:

$$A = ((2, A), (0, E), (4, I), (2, O), (7, U))$$

2.1 Concept Check

What is the runtime of Counting Sort?

What particular advantage does Counting Sort provide that the other sorting algorithms we have seen thus far not provide?

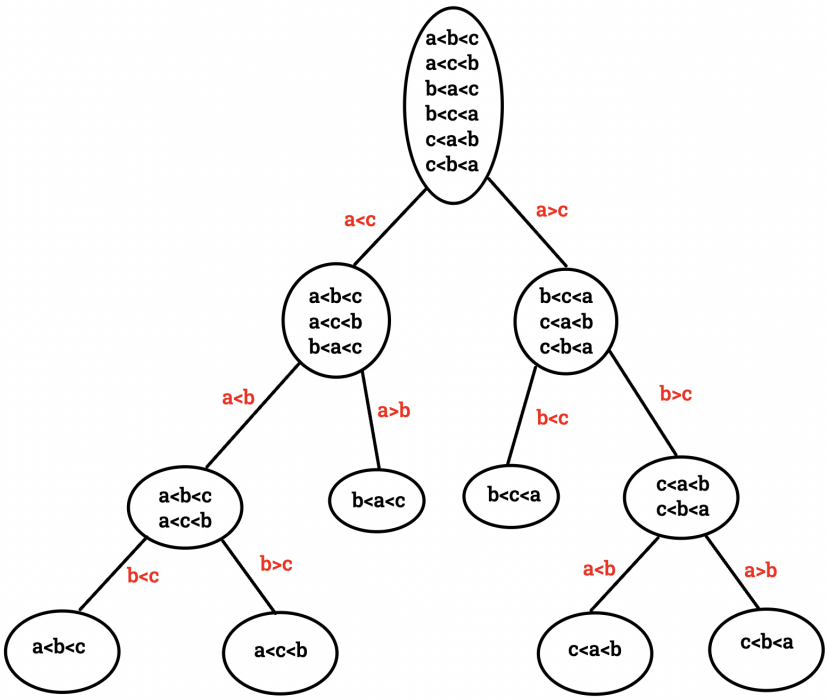
3 Comparison-Based Sorts

Many of the sorting algorithms you've seen before (MergeSort, QuickSort, Bubble Sort) are **comparison-based sorting algorithms**, which means they only make comparisons between the keys. But we've now seen CountingSort, which does *better* than our lower bound for comparison-based sorting algorithms in some cases.

Question 3.1. What does CountingSort do to disqualify it from being comparison-based?

For a comparison-based sorting algorithm, we can prove an *unconditional lower bound* on its runtime - any correct algorithm must run in time $\Omega(n \log n)$ on inputs of size n. To prove this, we model a comparison-based algorithm as making a binary tree of comparisons where edges of the tree are comparisons made and the vertices of the tree are the sets of orders that are still possible at that vertex (i.e. some orders are no longer possible at a vertex since they would contradict a comparison already made in the path from the root to that vertex). This means root vertices contain the set of all possible orderings of the input and leaf vertices represent a valid ordering of our input.

We note that each leaf has at most one ordering. If there is more than one ordering in a vertex, then there is more than one ordering possible, so the algorithm would need to make at least one more comparison to determine which ordering to return. In that case, the vertex could not be a leaf. Since there are leaves for all orderings and at most one ordering per leaf, then there are at least $n!$ leaves. Since the tree is binary, then there are at least $\log(n!) = \Theta(n \log n)$ levels of the tree. Given that each edge represents a comparison made, the number of edges in a path from the root vertex to a leaf vertex is the number of comparisons made for a comparisons-based sorting algorithm. Thus, a comparisons-based sorting algorithm must run in time $\Omega(n \log n)$ on inputs of size n.



Now suppose we allow *ternary* comparisons, where the algorithm can simultaneously query for the relative order of K_i, K_j, K_k for any i, j, k.

Question 3.2. Prove that in the ternary-comparison model, any valid sorting algorithm takes time $\Omega(n \log n)$. How does this runtime scale as we allow d-ary runtime?

This illustrates an important phenomenon - for *every* fixed value of d, we obtain an $\Omega(n \log n)$ lower bound, but if d is allowed to be a function of n this is not the case! Thus, it's important to specify which aspects of the model or analysis are held constant, and which grow with the input size.

4 Evaluating Algorithms

Recall the definitions of *computational problem* and *algorithm* from lecture (Section 8).

Definition 4.1. 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 solutions
- For each $x \in \mathcal{I}$, $f(x) \subseteq \mathcal{O}$

Definition 4.2. 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 $A(x) = \perp$.

Question 4.3. Which algorithm do you expect to be more efficient, Exhaustive-Search Sort, Insertion Sort, or Merge Sort? Before you answer this question, try running each algorithm on $A = (5, 2, 7, 10, 6, 13, 20, 1)$.

Question 4.4. Let $\Pi = (\mathcal{I}, \mathcal{O}, f)$ and $\Pi' = (\mathcal{I}', \mathcal{O}, f)$ be two computational problems such that $\mathcal{I} \subseteq \mathcal{I}'$. Does it follow that every algorithm A that solves Π also solves Π'? What about the converse? Justify your answers with proofs or counterexamples.

5 Reductions

Reductions are one of the most powerful tools we have as computer scientists. We'll talk more about reductions later in the course, but for now, here's a preview:

Informally, we say we can reduce problem A to problem B if we find ourselves saying: "if only I could solve problem B! Then, I could solve problem A pretty easily (with only a bit more time)." You can think of this as problem B being inside A's "belly", and A is able to use B as a subroutine for as many times as needed. We will need to keep track of the number of times that we call B and know the runtime of B, but you can always assume the correctness of B and treat it as a black-box.

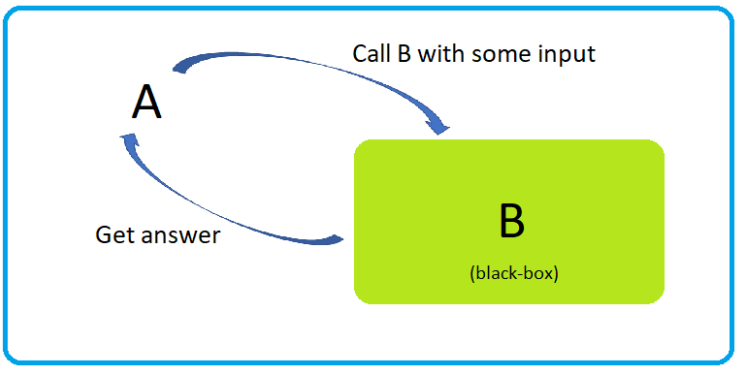


Figure 1: Illustration of a reduction from A to B. You can think of B as an oracle.

We can reason about reductions by pretending we have an oracle for problem B: some magic black-box function that instantly provides an answer to any possible input to B. Then, we can think of our reduction as a program for solving A that calls B's oracle as a subroutine. Here's an example, where we reduce the problem of getting someone's bank balance to that of getting their bank account password:

```
def get_bank_balance(username, encrypted_bank_info):
    password = ORACLE_get_bank_password(username)
    bank_info = encrypted_bank_info.decrypt(username, password)
```

```
balance = 0
for transaction in bank_info:
    balance += transaction.amount
return balance
```

Here, we say `get_bank_balance` **reduces to** `ORACLE_get_bank_password` (it's very easy to get wrong the direction of the reduction, so be careful!). Getting the balance of someone's bank account given only their username is a hard problem, and so is getting their bank account password. But we can imagine all the steps we would need to get their balance IF we had some way of getting their password; we could do it in only $O(n)$ more steps. So, if we call the runtime of `ORACLE_get_bank_password` $= RT(ORACLE)$, we can be sure that the runtime of `get_bank_balance` is $O(n + RT(ORACLE))$. In general, if your transformation takes $O(f(n))$ takes, we call it an $O(f(n))$-time reduction.

Definition 5.1. 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").

Notice that reductions are useful both in a "positive" and a "negative" sense: Let Π and Γ be computational problems such that $\Pi \leq \Gamma$. Then:

1. (Positive.) If there exists an algorithm solving Γ, then there exists an algorithm solving Π.
2. (Negative.) If there does not exist an algorithm solving Π, then there does not exist an algorithm solving Γ.

This is also why we have to be very careful when writing the **direction** of the reduction.

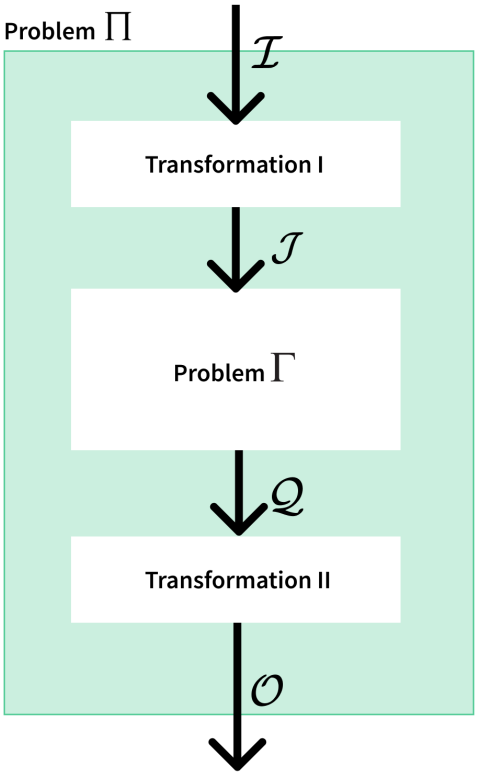


Figure 2: A reduction from Π to Γ which calls the oracle for Γ exactly once.

Now, recall the *IntervalSchedule* problem from Lecture 4, and think about the following questions:

Question 5.2. Conceptual questions about reduction

- Did we reduce Sorting to IntervalSchedule, or IntervalSchedule to Sorting? Why?

- What was the runtime of the transformation to/from the Sorting problem? How did we use this fact to bound the runtime of IntervalSchedule?

Now, let's try a reduction problem in full.

Question 5.3. Describe an algorithm to find the median of an array by a reduction to sorting.

Question 5.4. Given points $(x_0, y_0), \ldots, (x_{n-1}, y_{n-1})$ in the $\mathbb{R}^2$ plane, we want to determine whether point $p = (x_0, y_0)$ is *collinear* with at least three other points in the set.

- Describe a brute force algorithm that solves this problem. What is its time complexity?

- Describe a faster algorithm via a reduction to sorting. What is its time complexity? *(Hint: Points that are collinear with point p all have the same slope relative to p. For example, if* $p = (0, 1)$*, points* $(1, 2)$*,* $(2, 3)$*, and* $(107, 108)$ *all have slope* 1 *relative to p.)*