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

Section 1: Asymptotic Analysis, Sorting, Computational Problems

Harvard SEAS - Fall 2022 *Sept. 8, 2022*

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 $f(n) + g(n) = \Theta(f(n))$.

Question 1.3. Given the following functions

1. $f_1 = \sin(x)$
2. $f_2 = x\sin(x)$
3. $f_3 = x\log(x)$
4. $f_4 = x^2$
5. $f_5 = x^x$
6. $f_6 = 2^x$

Order the functions by time complexity. For instance, $f_a = O(f_b), f_b = O(f_c), \ldots f_y = O(f_z)$.

2 Counting Sort Review

We present Counting Sort here for item-key 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
Initialize an array C of length U, such that each entry of C is the start of an empty linked list.
foreach $i = 0, \ldots, n-1$ **do**
  Append (K_i, V_i) to the linked list $C[K_i]$.
Form an array A that contains the elements of $C[0]$, followed by the elements of $C[1]$, followed by the elements of $C[3]$, ...
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 item-key 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:

$$A = ((1, A), (0, E), (4, I), (2, O), (7, U))$$

2.1 Concept Check

What is the runtime of Counting Sort?

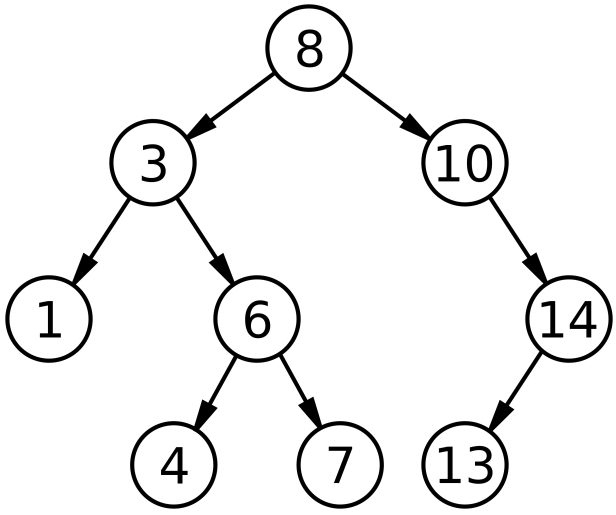
What particular advantage does Counting Sort provide that the other sorting algorithms we have seen thus far not provide?

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. For instance, one potential algorithm could compares each key to a node in the tree, and adds a left child if it is smaller or right child if it is greater. This algorithm makes this set of comparisons on the unsorted input list $\{8, 3, 6, 4, 7, 10, 14, 13, 1\}$:



Question 3.2. When evaluated on inputs $\{3, 4, 1, 2\}$ and $\{1, 2, 3, 4\}$, what comparisons does the above algorithm make and what leaf do we reach? What is wrong here?

We then prove that a comparison based sorting algorithm must have at least $n!$ leaves, since if two distinct permutations of $[n]$ reach the same leaf, any permutation we output as our sort must fail to sort at least one of them.

Question 3.3. Prove the above claim.

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 3.4. 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 phenomena - 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. We can have multiple algorithms that return the correct output. We typically distinguish between algorithms by comparing their efficiency.

- Given input $x = ((a, 3), (b, 5), (c, 2), (d, 5))$ for a sorting problem, what is $f(x)$?
- Explain how `Exhaustive-Search Sort`, `Insertion Sort`, and `Merge Sort` relate to $f(x)$.

- Is function f specified by a computational problem or by an algorithm?

Question 4.4. Which algorithm do you expect to be more efficient, Exhaustive-Search Sort, Insertion Sort, or Merge Sort? We will formalize this next week. Before you answer this question, try running each algorithm on $A = (5, 2, 7, 10, 6, 13, 20, 1)$.

Question 4.5. 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.