Limitations of Algorithm Power

11.1- 11.2

First, a brief homework review

5.1: 1abcd, 2abc

5.2: 5ab

6.1: 1, 4

6.2: 1, 3

6.6: 1, 11

7: Given n friends, each one can remain single or can be paired up with some other friend. Each friend can be paired only once. Find out the total number of ways in which friends can remain single or can be paired up.

9.1: 9, 11

9.2: 1, 4

9.3: 2

9.4: 1

10.1: 2

10.2: 7

First, a brief homework review

Group work to discuss any questions on homework. Let’s get those portfolios up to date!

Limitations of Algorithm Power

We have talked about many algorithms this semester- they are very powerful

But the power of algorithms is not unlimited

So far in this class we've been developing algorithms and data structures to solve certain problems as quickly as possible. Starting with this lecture, we'll turn the tables, by proving that certain problems cannot be solved as quickly as we might like them to be.

Chapter outline

11.1: discusses methods for obtaining lower bounds, which are estimates on a minimum amount of work needed to solve a problem.

11.2: discusses decision trees. This technique allows us, among other applications, to establish lower bounds on the efficiency of comparison-based algorithms for sorting and for searching in sorted arrays.

11.3: deals with the question of intractability: which problems can and cannot be solved in polynomial time.

11.4: deals with numerical analysis. This branch of computer science concerns algorithms for solving problems of “continuous” mathematics—solving equations and systems of equations, evaluating such functions as sin x and ln x, computing integrals, and so on

Lower bound arguments

To measure the efficiency of an algorithm:

- We can establish an algorithm’s asymptotic efficiency class (say, for the worst case) and see where this class stands with respect to the hierarchy of efficiency classes outlined in Section 2.2.
- A “fairer” approach is to ask how efficient a particular algorithm is with respect to other algorithms for the same problem

Lower bound arguments

Selection sort seems to be a fast performing algorithm with a quadratic run time, but when compared to other sorting algorithms with O(n log n) run times, it is considered more slow.

Lower bound arguments

When we want to understand the efficiency of an algorithm with respect to other algorithms for the same problem, we should look at the best possible efficiency any algorithm solving the problem may have.

Lower bound arguments

When we want to understand the efficiency of an algorithm with respect to other algorithms for the same problem, we should look at the best possible efficiency any algorithm solving the problem may have.

Knowing a lower bound can tell us how much improvement we can hope to achieve in our search for a better algorithm for the problem in question.

Lower bound theory

According to the lower bound theory, for a lower bound L(n) of an algorithm, it is not possible to have any other algorithm (for a common problem) whose time complexity is less than L(n) for random input. Also, every algorithm must take at least L(n) time in the worst case.

Lower Bound Theory

The Lower Bound is very important for any algorithm. Once we calculated it, then we can compare it with the actual complexity of the algorithm and if their order is the same then we can declare our algorithm as optimal.

In short..

Lower bound: an estimate on a minimum amount of work needed to solve a given problem

A lower bound on a problem is a big-Omega bound on the worst-case running time of any algorithm that solves the problem

Lower bound can be:

- an exact count
- an efficiency class (Ω)

Examples

- Number of comparisons needed to find the largest element in a set of n numbers
- Number of comparisons needed to sort an array of size n
- Number of comparisons necessary for searching in a sorted array
- Number of multiplications needed to multiply two n-by-n matrices

Tight lower bounds

Tight lower bound: there exists an algorithm with the same efficiency as the lower bound

Problem	Lower Bound	Tightness
Sorting	Ω(nlogn)	yes
Searching a sorted array	Ω(logn)	yes
Element Uniqueness	Ω(nlogn)	yes

Methods for establishing lower bounds

1. Trivial lower bounds
2. Information-Theoretic arguments (decision trees)
3. Adversary arguments
4. Problem reduction

Trivial lower bounds

The simplest method of obtaining a lower-bound class is based on counting the number of items in the problem’s input that must be processed and the number of output items that need to be produced.

Trivial lower bounds

The simplest method of obtaining a lower-bound class is based on counting the number of items in the problem’s input that must be processed and the number of output items that need to be produced.

Trivial lower bounds: based on counting the number of items that must be processed in input and generated as output

Trivial lower bounds

The simplest method of obtaining a lower-bound class is based on counting the number of items in the problem's input that must be processed and the number of output items that need to be produced.

Trivial lower bounds: based on counting the number of items that must be processed in input and generated as output

It is the easiest method to find the lower bound. The Lower bounds which can be easily observed on the basis of the number of input taken and the number of output produced are called Trivial Lower Bound.

Trivial lower bounds

The difficulty with this technique is that it often suggests a lower bound that it is too low.

For example, the trivial lower bound for the complexity of comparison-based sorts is Ω(n), because the algorithm must read the array and output the results. This bound is too low.

Trivial lower bounds

Another difficult is that algorithms do not always have to read all the inputs to solve the problem.
Consider the problem of searching for a key in an array. The augment that the algorithm must read the entire array suggests a complexity of $\Omega(n)$, but if the array is already sorted, then the algorithm does not have to read all of the array to find the key.

Decision Trees

A decision tree is, as the name suggests, a tree:

- Each internal node in the tree is labeled by a query, which is just a question about the input.
- The edges out of a node correspond to the possible answers to that node's query.
- Each leaf of the tree is labeled with an output.

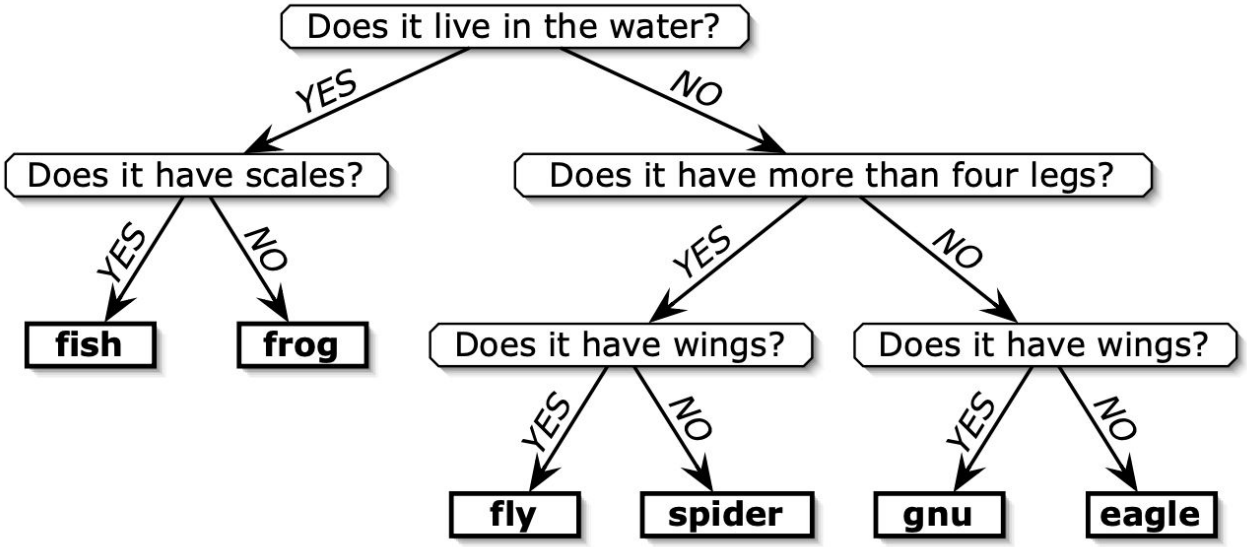
To compute with a decision tree, we start at the root and follow a path down to a leaf. At each internal node, the answer to the query tells us which node to visit next. When we reach a leaf, we output its label.

Decision Trees

For example, the guessing game where one person thinks of an animal and the other person tries to figure it out with a series of yes/no questions can be modeled as a decision tree. Each internal node is labeled with a question and has two edges labeled ‘yes’ and ‘no’. Each leaf is labeled with an animal.

Decision Trees

The guessing game



A decision tree to choose one of six animals.

Decision Trees

We define the running time of a decision tree algorithm for a given input to be the number of queries in the path from the root to the leaf.

For example, in the ‘Guess the animal’ tree on the previous slide,T(frog) = 2.

Decision Trees

Thus, the worst-case running time of the algorithm is just the depth of the tree.

This definition ignores other kinds of operations that the algorithm might perform that have nothing to do with the queries.

But the number of decisions is certainly a lower bound on the actual running time, which is good enough to prove a lower bound on the complexity of a problem.

Decision Trees

The guessing game describes a binary decision tree, where every query has only two answers.

We may sometimes want to consider decision trees with higher degree. For example, we might use queries like ‘Is x greater than, equal to, or less than y?’ or ‘Are these three points in clockwise order, collinear, or in counterclockwise order?’

A k-ary decision tree is one where every query has (at most) k different answers.

Decision Trees

Information Theory- Most lower bounds for decision trees are based on the following simple observation: The answers to the queries must give you enough information to specify any possible output.

If a problem has N different outputs, then obviously any decision tree must have at least N leaves. (It's possible for several leaves to specify the same output.)

Thus, if every query has at most k possible answers, then the depth of the decision tree must be at least $[\log_k N] = \Omega(\log N)$.

Decision Trees, sorting

Now let’s consider the classical sorting problem — Given an array of n numbers, arrange them in increasing order. Unfortunately, decision trees don’t have any way of describing moving data around, so we have to rephrase the question slightly:

Decision Trees, sorting

Now let’s consider the classical sorting problem — Given an array of n numbers, arrange them in increasing order. Unfortunately, decision trees don’t have any way of describing moving data around, so we have to rephrase the question slightly:

Given a sequence $\langle x1 , x2 , \ldots , xn \rangle$ of n distinct numbers, find the permutation π such that $x\pi(1) < x\pi(2) < \cdots < x\pi(n)$.

Decision Trees, sorting

Now a k-ary decision-tree lower bound is immediate. Since there are n! possible permutations π, any decision tree for sorting must have at least n! leaves, and so must have depth Ω(log(n!)). To simplify the lower bound, we apply Stirling's approximation

$$n! = \left(\frac{n}{e}\right)^n \sqrt{2\pi n}\left(1 + \Theta\left(\frac{1}{n}\right)\right) > \left(\frac{n}{e}\right)^n.$$

Decision Trees, sorting

This gives us the lower bound

$$\lceil \log_k(n!) \rceil > \left\lceil \log_k \left(\frac{n}{e}\right)^n \right\rceil = \lceil n \log_k n - n \log_k e \rceil = \Omega(n \log n).$$

This matches the O(n log n) upper bound that we get from mergesort, heapsort, or quicksort, so those algorithms are optimal. The decision-tree complexity of sorting is Θ(n log n).

Adversary Arguments

You can think of the adversary lower bound technique as devising a strategy to construct a worst case input for an unknown correct algorithm to solve problem P.

We will view this process as a game between an algorithm A and an adversary (or devil) D. We assume that D has unlimited computational power.

Adversary Arguments

The goal of algorithm A is to minimize the number of rounds until its computation to solve P is completed.

The goal of the adversary D is to maximize the number of rounds until A could be done.

Adversary Arguments

As a simple example, let’s consider playing the game of “20 questions” against the adversary D.

In this game, D picks an integer x between 1 and n, and the algorithm A must determine x by asking questions (to D) of the form “Is the number you have picked less than y?”

Adversary Arguments

We now argue that D can force A to ask at least $[\log_2 n]$ questions before A can be certain about the value for x.

We call the way in which the adversary D plays this game the adversary strategy to be sure we do not confuse it with the algorithm A is using to determine x.

The adversary is allowed to keep changing his mind about x but must answer in a consistent manner. That is, at the end of the game, the adversary must be able to give a value for x that is consistent with all answers given throughout the game.

So it is possible that D did indeed have x in mind from the start.

Adversary Arguments

Here is an adversary strategy that leads to the stated lower bound.

The adversary maintains a list L of all possible values that are still legal for x. So initially the n integers {1, 2, 3, . . . , n − 1, n} are placed in L.

Adversary Arguments

Each time A asks D if $x < y$, D counts how many of of the integers in L are greater than x.

If at least half of the numbers in L are greater than x then D will respond “yes,” and otherwise D will respond “no.”

If D responds “yes” (i.e., $x < y$) then all elements in the list that are $\geq y$ must be removed from L (or otherwise the adversary would be lying).

Likewise, if D responds “no” (i.e., $x \geq y$) then all elements in the list that are $< y$ must be removed from L.

Adversary Arguments

Observe that a correct algorithm A cannot know the value of x (and thus not finish executing) until L contains only a single item. (If there are two or more items in L then whatever A outputs could be the wrong one.)

Adversary Arguments

We must now determine how many rounds the adversary D can force before L could possibly reach size 1. Initially |L| = n. Now let's consider any single round of this game and let |L| = s. Since D responds in a way that least half of the items are consistent with the response, at the next round |L| ≥ [s/2].

From this it follows that i = $\log_2 n$ is the smallest possible value for i for which |Li | = 1. (This could be proven by induction.)

Adversary Arguments

Let’s do an example to illustrate this where n = 100.

$$|L_0| = 100,\ |L_1| \geq 50,\ |L_2| \geq 25,\ |L_3| \geq 13,\ |L_4| \geq 7,\ |L_5| \geq 4,\ |L_6| \geq 2,\ |L_7| \geq 1$$

Adversary Arguments

After 6 rounds A could not be done since there must be at least two values for x that are consistent with all answers given so far. So, for this problem when n = 100, the best any algorithm A can do is to make 7 questions.

Notice that we have not placed any restrictions at all on A. We have just shown that regardless of the algorithm A, the adversary strategy described above guarantees that A could not possibly be done until at least 7 rounds.

Problem Reduction

We have already talked about the problem-reduction approach in Section 6.6.

There, we discussed getting an algorithm for problem P by reducing it to another problem Q solvable with a known algorithm.

A similar reduction idea can be used for finding a lower bound. To show that problem P is at least as hard as another problem Q with a known lower bound, we need to reduce Q to P (not P to Q!).

So any algorithm solving P would solve Q as well. Then a lower bound for Q will be a lower bound for P.

Homework Problem!

Please complete 11.1: 8