Class Notes

CIS 502 Analysis of Algorihtm

Da Kuang

University of Pennsylvania

Part I

Algorithm Basics

1 Introduction

First, let's talk about cases. A case of input for an algorithm is associated with an instance of a problem. For the sorting problem (where we want to find a permutation of a set in a specific order), I can look at an instance like the set of numbers {1, 5, 4, 2, 6}. This set of numbers would be the input to a sorting algorithm that purports to solve the sorting problem, like Selection Sort, or one of the other sorting algorithms out there.

The same sets of inputs can be given to any algorithm that wants to solve a problem. It doesn't matter what sorting algorithm I use, the set of inputs is always the same (because, by definition, they're all instances of the same problem). However, a given case can be better or worse for a given algorithm. Some algorithms always perform the same no matter what the inputs are, but some algorithms might do worse on some inputs. However, this means that every algorithm has some best case and some worst case; we also sometimes talk about the average case (by taking the average of all the cases) or the expected case (when we have some reason to expect that one case will be more common than others).

1.1 Algorithm Case Examples

The problem of "find the minimum of an unsorted list" always works the same for every possible input. No matter what clever algorithm you write, you have to check every element. It doesn't matter if you have a list of zeros or a list of random numbers or a list where the first element is the minimum, you don't know until you get to the end. Every case is the same for that algorithm, so the best case is the worst case, and also the average case and the expected case. If the list was sorted, we could do better, but that's a different problem.

The problem of "find a given element in a list" is different. Assuming you were using an algorithm that does a linear walk through the list, it might turn out that the given element was the first element of the list and you're done immediately. However, it might also be the last element of the list, in which case you have to walk the whole thing before you find it. So there you had a best case and a worst case.

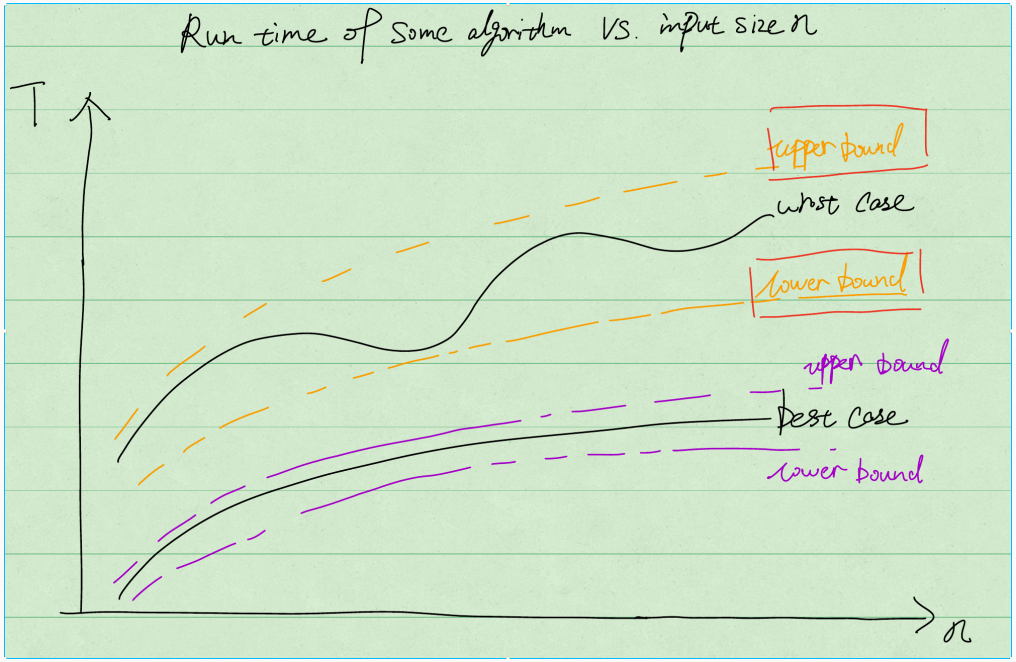
1.2 Algorithms as Functions of Input Size

When we want to analyze an algorithm, us algorists think about every possible case we could throw at the algorithm. Usually, the two most interesting cases are the best case and the worst case. If you think of the algorithms runtime as a function of its input, the best case is the input that minimizes the function and the worst case is the input that maximizes the function. I'm using "function" in the Algebra math sense here: a series of x/y pairs (input/output pairs, or in this case "input size/number of execution steps") that draw a line.

Because algorithms' run-time is a function of its input, we have a different best case (and worst case) for

each possible input size. So sometimes we treat the best case as a single input, but it's really a set of inputs (one for each input size). The best case and worst case are very concrete things with respect to a given algorithm.

1.3 Bounds



Now what about bounds? Bounds are functions that we use to compare against a given algorithm's function. There are an infinite number of boundary functions we could consider. How many possible kinds of lines can you draw on a graph? That's how many boundary functions there are. Most algorists are usually only interested in a few specific functions: things like the constant function, the linear function, the logarithmic function, the

exponential function, etc.

An upper bound is a function that sits on top of another function. A lower bound is a function that sits under the other function. When we talk about Big O and Big Omega, we don't care if the bounds are ALWAYS above or below the other function, just that after a certain point they always are (because sometimes algorithms get weird for small input sizes).

There are an infinite number of possible upper bounds for any given function, and an infinite number of possible lower bounds for any given function. But this is one of those weird times when we're talking about different sizes of infinities. To be an upper bound, the function must not be below the other function, so we rule out the infinite number of functions below the other function (so it's smaller than the set of all possible functions).

Of course, just because there are infinite upper bounds, doesn't mean they're all useful. The function f(∞) is an upper bound for every function, but that's like saying "I have less than an infinite number of dollars" - not particularly useful for figuring out if I'm penniless or a millionaire. So we are often interested in an upper bound that is "**tight**" (also known as a "least upper bound" or "supremum"), for which there is no better upper bound.

1.4 Best/Worst Case + Lower/Upper Bound

We have best/worst cases that represent the upper and lower functions of an algorithms' runtime function. We have upper and lower bounds that represent other functions that could be on top or below (respectively) any other function. They can be combined to articulate key ideas about algorithms.

- **Worst Case Lower Bound**: A function that is a boundary below the algorithms' runtime function, when that algorithm is given the inputs that maximize the algorithm's run time.
- **Worst Case Upper Bound**: A function that is a boundary above the algorithms' runtime function, when that algorithm is given the inputs that maximize the algorithm's run time.
- **Best Case Lower Bound**: A function that is a boundary below the algorithms' runtime function, when that algorithm is given the inputs that minimize the algorithm's run time.
- **Best Case Upper Bound**: A function that is a boundary above the algorithms' runtime function, when that algorithm is given the inputs that minimize the algorithm's run time.

1.5 Examples of Case Bounds

Let's give concrete examples of when we might care about each of these:

Worst Case Lower Bound

The classic example here is comparison-based sorting, which is famously known to be $\Omega(n \log(n))$ in the worst case. No matter what algorithm you devise, I can pick a set of worst-case inputs whereby the tightest lower bound function is log-linear. You cannot make an algorithm that beats that bound for the worst case, and you shouldn't bother trying. It's the basement of sorting. Of course, there are many lower bounds for the worst case: constant, linear, and sub-linear are all lower bounds. But they are not useful lower bounds, because there the log-linear lower bound is the tightest one.

Best Case Lower Bound

Insertion Sort works by walking through the list, and inserting any out-of-order it comes across in the right place. If the list is sorted, it will only need to walk through the list once without doing any inserts. This means that the tightest lower bound of the best case is $\Omega(n)$. You cannot do better than that without sacrificing correctness, because you still need to be able to walk through the list (linear time). However, the lower bound for the best case is better than the lower bound for the worst case!

Worst Case Upper Bound

We are often interested in finding a tight upper bound on the worst case, because then we know how poorly our algorithm can run in the worst of times. Insertion sort's worst case is a list that is completely out of order

(i.e. completely reversed from its correct order). Every time we see a new item, we have to move it to the start of the list, pushing all subsequent items forward (which is a linear time operation, and doing it a linear number of times leads to quadratic behavior). However, we still know that this insertion behavior will be $O(n^2)$ in the worst case, acting as a tight upper bound for the worst case. It's not great, but it's better than an upper bound of, say, exponential or factorial! Of course, those are valid upper bounds for the worst case, but again that's not as useful as knowing that quadratic is a tight upper bound.

Best Case Upper Bound

What's the worst our algorithm can do in the best of times? In example before of finding an element in a list, where the first element was our desired element, the upper bound is O(1). In the worst case it was linear, but in the best case, the worst that can happen is that it's still constant. This particular idea isn't usually as important as Worst Case Upper Bound, in my opinion, because **we're usually more concerned with dealing with the worst case**, not the best case.

Some of these examples are actually Θ, not just O and Ω. In other cases, I could have picked lower or upper bound functions that weren't tight, but were still approximate enough to be useful (remember, if we're not being tight, I have an infinite well to draw from!). Note that it can be difficult to find compelling examples of different case/bound combinations, because the combinations have different utility.

1.6 Misconceptions and Terminology

Frequently, you'll see people with misconceptions about these definitions. In fact, many perfectly good Computer Scientists will use these terms loosely and interchangeably. However, the idea of cases and bounds ARE distinct, and you would do well to make sure you understand them. Does this mean the difference will come up in your day-to-day? No. But when you are choosing between a few different algorithms, you want to read the fine print about the cases and bounds. *Someone telling you that their algorithm has a Best Case Upper Bound of O(1) is probably trying to pull the wool over your eyes - make sure you ask them what the Worst Case Upper Bound is!*

1.7 Reference

Upper bound vs lower bound for worst case running time of an algorithm

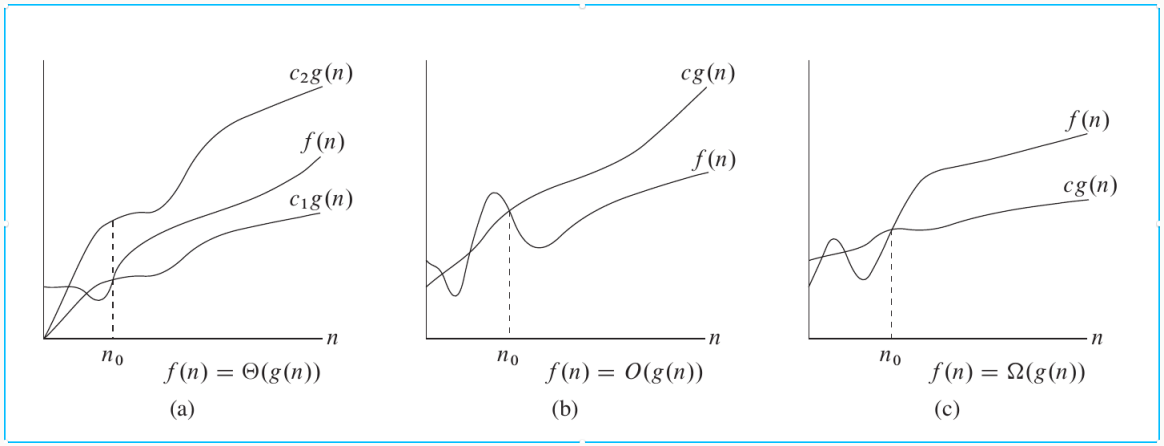
2 The asymptotic efficiency of algorithms

The order of growth of the running time of an algorithm gives a simple characterization of the algorithm's efficiency and also allows us to compare the relative performance of alternative algorithms. Although we can sometimes determine the exact running time of an algorithm, the extra precision is not usually worth the effort of computing it.

We are studying the asymptotic efficiency of algorithms. That is, we are concerned with how the running time of an algorithm increases with the size of the input in the limit, as the size of the input increases without bound. Usually, an algorithm that is asymptotically more efficient will be the best choice for all but very small inputs.

2.1 Asymptotic notation

In this class, we usually apply asymptotic notation to characterize the running times of algorithms. Even when we use asymptotic notation to apply to the running time of an algorithm, we need to understand which running time we mean. Sometimes we are interested in the **worst-case running time**. Often, however, we wish to characterize the running time no matter what the input. In other words, we often wish to make **a blanket statement that covers all inputs, not just the worst case**. We shall see asymptotic notations that are well suited to characterizing running times no matter what the input.



Θ-notation

$\Theta(g(n)) = f(n)$: there exist positive constant$c_1, c_2$and$n_0$such that$0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n)$ for all $n \geq n_0$

$g(n)$ is an asymptotically **tight bound** for $f(n)$.

O-notation (Worst of the worst)

$\Theta(g(n)) = f(n)$: there exist positive constant$c$and$n_0$such that$0 \leq f(n) \leq cg(n)$ for all $n \geq n_0$

$g(n)$ is an asymptotic **upper bound** for $f(n)$.

Using O-notation, we can often describe the running time of an algorithm merely by inspecting the algorithm's overall structure. Since O-notation describes an upper bound, when we use it to bound the worst-case running time of an algorithm, we have a bound on the running time of the algorithm on every input—the blanket statement we discussed earlier.

Notes, when we say "the running time of insertion sort is $O(n^2)$", we that "the **worst-case running time** is $O(n^2)$".

Ω-notation (best of the best)

$$\Omega(g(n)) = f(n) : \text{there exist positive constant} c \text{and} n_0 \text{such that} 0 \leq cg(n) \leq f(n) \text{ for all } n \geq n_0$$

$g(n)$ is an asymptotic **lower bound** for $f(n)$.

When we say that the running time (no modifier) of an algorithm is $\Omega(g(n))$, we mean that no matter what particular input of size n is chosen for each value of n, the running time on that input is at least a constant times $g(n)$, for sufficiently large n. Equivalently, we are giving a lower bound on **the best-case running time** of an algorithm.

2.2 Lower Bounds

Usually in *Algorithm Analysis* we are looking for upper bounds on cost, as in "Algorithm X solves problem Y in no more than $O(n^{22/7})$ time in the worst case." Upper bounds are useful when we want to advertise that our algorithm is good. But what if **we want to argue that no other algorithm can be better, or perhaps that some problem is so hard that we can't possibly hope to find a good solution to it**?

For this we need a lower bound. Lower bounds come in two flavors:

- A lower bound on **an algorithm** is just a **big-Omega** bound on its worst-case running time. "I don't know exactly how long Bogosort takes in general, but I can prove its worst-case time is $\Omega(n^2)$."
- A lower bound on **a problem** is a **big-Omega** bound on the worst-case running time of any algorithm that solves the problem:
  - "Any comparison-based sorting routine takes $\Omega(n \log n)$ time."
  - "There is an $\varepsilon > 0$ such that any algorithm for Independent Set takes $\Omega((1+\varepsilon)n)$ time."

2.3 Proving Lower Bounds

For any computational problem there are many different (often infinitely many) algorithms that solve the problem.

- **Upper bound of a problem**: To prove $O(f(n))$ for the problem, we need to show that there is an algorithm

that solves the problem and takes no more than $f(n)$ time on each input of length n.

- **Lower bound of a problem**: To prove $\Omega(g(n))$ for the problem, we need to show that for each algorithm A and for each (sufficiently large) length n, there is an input on which A takes at least $g(n)$ time.
- **Tight bounds of a problem**: When we can prove lower and upper bounds for a problem that are the same, then we say that we have tight bounds.

Proofs of lower bounds are often harder since we have to reason about every possible algorithm to solve the problem. In order to prove lower bounds, we often have to restrict the power of the algorithms we consider. This is done by specifying a model of computation, which lists the primitive operations that an algorithm is allowed to perform. Most of the lower bounds developed in the literature are fairly ad hoc but there are a few general techniques that are used.

Why one might want to prove lower bounds:

- From the theoretical point of view we can say that we understand a computational problem really well if we can prove tight lower and upper bounds and understanding of these problems is one of the major goals of computer science.
- More practically, tells us where to stop in our search for better and better algorithms for a problem.
- The lower bound proof often highlights good and bad strategies for algorithms to follow and thereby might help in the design of good algorithms.

Information-Theoretic Techniques

These techniques work in **restricted models** of computation, such as the **comparison tree model**. In this model, an algorithm is only allowed to perform comparisons between its input elements. The structure of any such algorithm can be viewed as a tree; hence the name of the model.

Each internal node is labeled by a comparison and the two children of an internal node represent the two possible results of the comparison.

The leaves represent solutions to the problem. If we can lower bound the number of possible distinct solutions, this is also a lower bound on the number of leaves. This in turn translates into a lower bound on the height of the comparison tree. Since the height of the tree represents the worst-case number of comparisons performed by the algorithm we get the desired lower bound.

Potential functions

This technique for proving lower bounds associates some sort of potential function that needs to be changed from some initial value to some final value by the algorithm. If we can show that each step of the algorithm does not change the potential too much, we can get a lower bound on the number of steps.

Adversary lower bounds

One problem that we have with proving lower bounds is that different algorithms have different worst-cases. Thus we cannot set up one input and analyze all algorithms on this input and get a good lower bound. The two techniques we have discussed so far do not create bad inputs tailored to each algorithm to prove lower bounds. The adversary technique does. How can we demonstrate a worst-case input for each of a potentially infinite collection of algorithms?

The trick is to design a strategy for the adversary that fields questions made by any algorithm and answers these questions in such a way that the progress of the algorithm to the solution is slowed down as much as possible. A strategy for an adversary defines an answer down as much as possible. A strategy for an adversary defines an answer for each possible question in each possible 'state', where a state is the set of questions that have been asked and answers that have been given to these questions.

Thus it can be seen that **the adversary is adaptively creating a worst-case input for the algorithm**. ***One constraint is that the adversary should give a consistent set of answers***, i.e., it should be possible to create an input for which all of the adversary's answers are correct. An analysis of the adversary's strategy can then lead to a lower-bound for all comparison-based algorithms for a problem.

Why does worst-case analysis help us to find the lower bound, which is the best case?

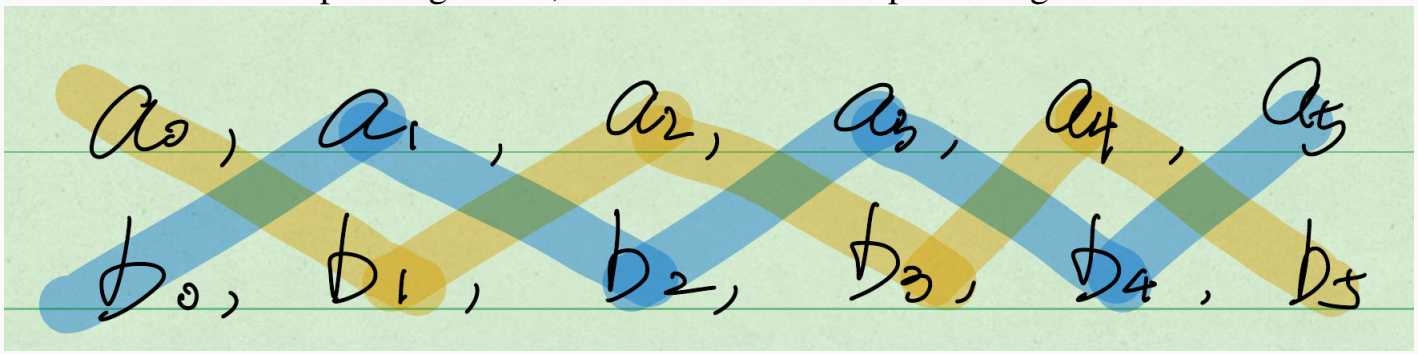
Looking at the adversary strategy gives us a good idea of how to design an optimal algorithm for the problem. It provides us with a way to create worst-case inputs for any algorithm we consider and it also points

to what kinds of comparisons are good to make.

Example:

- Finding the Maximum and the Minimum of n elements
- Merging two sorted lists of length
  - Let $a_1 < \cdots < a_n$ and $b_1 < \cdots < b_n$ be the two lists of length n. When the algorithm compares a_i with b_j, the adversary answers $a_i < b_j$ whenever $i \leq j$ and $b_j < a_i$ otherwise.
  - We claim that this strategy ensures that the algorithm must perform $2n - 1$ comparisons.

    Proof. To prove the claim we make a more specific claim. For $i = 1, \cdots, n-1$ the algorithm must compare b_i with a_i as well as a_{i+1}. Also, the algorithm must compare b_n with a_n.
    The aim of the adversary is to make the merging algorithm running as slow as possible. The given adversary strategy makes two interleaving ascending arrays and makes sure that b_i is only able to be placed before or after a_i such which slows down the merging procedure. Given this adversary strategy, even for the best/simplest algorithm, it still cost $2n - 1$ steps to merge two list.



This specific claim lists 2n − 1 comparisons and hence would prove the lower bound. But the adversary should give a consistent set of answers.

- *[!] The algorithm which compare b_i with a_i as well as a_{i+1}. Is rather intuitive, how does it come? Why we have the confident to say it is the lower bound, in other words, best of the best?*

  Proof. Suppose the algorithm fails to perform the comparison b_i with a_i for some i. Then both of the following orders are consistent with all of the adversary answers:

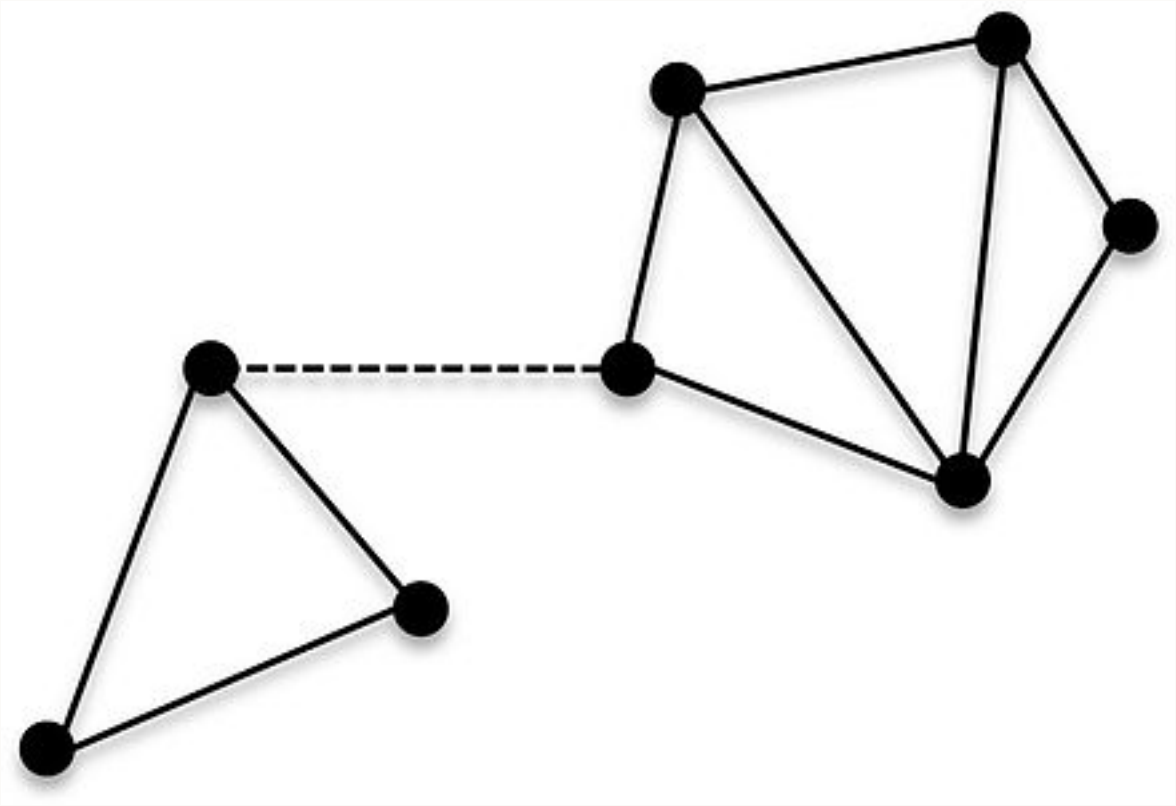
$$a_1 < b_1 < a_2 < \cdots < a_{i-1} < b_{i-1} < \boldsymbol{a_i < b_i} < \cdots < b_n$$

$$a_1 < b_1 < a_2 < \cdots < a_{i-1} < b_{i-1} < \boldsymbol{b_i < a_i} < \cdots < b_n$$

  Thus the algorithm cannot know which of these two orders is correct and hence cannot stop. Similarly we can show that if any comparison of b_i with a_{i+1} is not performed by the algorithm, again there are at least 2 possible orderings consistent with all the answers obtained so far.

- You may have noticed that although we have couched the last argument as an adversary argument, the adversary does not really adapt to the questions asked by the algorithm. The worst-case input that she produces is the same for all algorithms! Nevertheless, the adversary argument provides greater clarity especially emphasizing the point that if the algorithm decides to output one of two surviving orderings at the end, the adversary can always arrange for the correct order to be the other one.

Connectivity An undirected graph is connected when it has at least one vertex and there is a path between every pair of vertices. Equivalently, a graph is connected when it has exactly one connected component. In a connected graph, there are no unreachable vertices. An undirected graph that is not connected is called disconnected. An undirected graph G is therefore disconnected if there exist two vertices in G such that no path in G has these vertices as endpoints. The below graph becomes disconnected when the dashed edge is removed.



Adversary Lower Bounds in the Edge Probe Model You are given the vertex set V of a graph, but its edges are initially unknown. In the edge probe model the algorithm is allowed to query any pair of vertices (i, j) and learns whether there is an edge between i and j. The algorithm needs to determine whether the graph has some property.

For example, let us imagine that the goal of the algorithm is to determine **if the graph is connected**. The model of computation here counts the number of edge probes made by the algorithm. A **trivial upper bound** is $\binom{n}{2}$, since the algorithm can query all pairs, determine the entire graph, and then easily decide if it is connected.

To give a lower bound, we imagine an adversary answering the probes made by the algorithm and design a strategy for the adversary to answer these probes to make the algorithm take as many steps as possible. A naive first attempt might be to suggest that the adversary keeps saying "no" to each edge probe. But then the algorithm that first probes all $n - 1$ pairs involving the vertex 1, would conclude in $n - 1$ probes that the graph is not connected, because vertex 1 is isolated.

Likewise answering "yes" to every edge probe does not work since an algorithm can find a spanning tree in just n – 1 probes. Thus, in order to create a difficult graph the adversary must judiciously mix "yes" and "no" answers. A good adversary strategy is the following:

Playing against any algorithm, the adversary maintains the connected components in the graph formed by the edges the algorithm has discovered already. Now when the algorithm makes the probe (i, j)*, the adversary answers "yes" if and only if* (i, j) *is the last possible edge connecting the component of* i *with the component of* j*.*

In other words, if A says "Yes, there is an edge between i and j", it means *i is the only vertex connects to j and A has checked every vertex with j except i.*

If the algorithm does not make all $\binom{n}{2}$ probes, the graph will be disconnected, with the possibility of becoming connected if the unprobed pairs are revealed to be edges. Thus the algorithm will be unable to stop and correctly determine if the graph is connected. How do we prove this? We do so by proving the following statements:

Lemma 2.1 *For any algorithm A, at any stage, let C be a connected component that has been revealed by the probes that A has made. Then A must have probed every pair of distinct vertices in C, which cost* $\binom{n}{2}$ *steps.*

Proof. *We prove checking connectivity cost* $\binom{n}{2}$ *steps by induction.*

Base case, the number of probe has been executed is 0. Initially no edges have been discovered, each component is a singleton vertex, and it is vacuously true that every pair of distinct vertices in any component has been probed.

Inductive hypothesis, suppose after the algorithm makes its i-th probe, the graph is connected.

We want to show that the graph is still connected after the (i+1)-th probe. Say this probe is the pair (u, v). If this probe gets the answer "no", then no edges are added and the connected components remain the same. Hence the inductive hypothesis "carries over" and all pairs in each component have been probed as before.

If on the other hand, suppose the pair (u, v) is revealed to be an edge. Now a new connected component is

formed merging together the components of u and v. Have all pairs of vertices in this component been probed? By the inductive hypothesis every pair within the component of u and within the component of v (before edge (u, v) was added) must have been probed. By the adversary strategy, all pairs between vertices in these two components must have already been probed for the adversary to answer yes. Thus all pairs within the new component containing u and v must have been probed, and the inductive proof is complete.

Lemma 2.2 *If the algorithm proceeds to probe all pairs, it will discover that the graph is connected.*

Proof. This is easy to see from the adversary strategy. The aim of the adversary is, suppose there is a connected graph with n vertex, trying to make the graph so that edge detecting algorithm could discover the connectivity as late as possible.

2.4 Conclusions

This handout presents a far rosier picture of lower bound theory than is true. In fact non-trivial lower bounds are practically unknown on general computational models like the computers you use. The best lower bounds are obtained when restricted models of computing such as the comparison tree model are considered. Even here there is not an abundance of general techniques. Proving a lower bound for a particular problem usually calls for a good deal of ingenuity and perseverance. The edge probe model has been widely studied. Properties such as connectivity that require an algorithm to probe all possible pairs are said to be **evasive**. It turns out that many interesting properties are evasive, and there are many beautiful results that show that a large

class of properties are evasive.

Part II

Algorithm Design and Analysis Paradigms

3 Recurrence Relation

4 Quick Sort

- Input: Array of $n$integers.
- Outputs: These integers in certain order.

4.1 Algorithm:

- Pick an element $x$as the pivot element
- By comparing each of the elements with the pivot, place it in one of the two sets $S = y : y < x$, $L = y : y > x$.

- Recursively sort S and L.
- Output $S \times L$.

4.2 Performance:

- Worst case:
  - Each partition routine produces one sub-problem with n-1 elements and one with 0 elements.
    - $T(n) \geq \Theta(n) + T(n-1)$
  - Based on substitution method
    - $T(n) = \Theta(n^2)$
- Expected worst case behavior of an algorithm: algorithm which is randomized.
- Expected number of comparison made by QuickSort in the worst case.
- Algorithm: pick pivot uniformly of random from the elements to be sorted.
  - Let x be the total amount of comparison are formed by QuickSort.
  - Let x_{ij} be a random variable that is :

$$x_{ij} = \begin{cases} 1 & \text{if i-th smallest element and j-th smallest element are compared.} \\ 0 & \text{otherwise.} \end{cases}$$

  - $x = \sum_{i<j} x_{ij}$
  - $E[x] = E[\sum_{i<j} x_{ij}] = \sum_{i<j} E[x_{ij}]$ (Linearity of Expectation) $= \sum_{i<j} Pr[x_{ij} = 1]$

- x_i and x_j are compared if and only if the firs element to be chosen as a pivot from x_{ij} is either x_i or x_j.
- $\Pr\{z_i \text{ is compared to } z_j\} = \frac{2}{j-i+1}$

- $E[x] = \sum_{i<j} Pr[x_{ij} = 1] = \sum_{i<j} \frac{2}{j-1+1} = \sum_{i<j} \frac{2}{k+1} < \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \frac{2}{k} = \sum_{i=1}^{n-1} O(\log n) = O(n \log n)$

5 Selection/ order

- Input: A set of A of n integers and a rank i, with $1 \geq i \geq n$.
- output: The element $x \in A$ that is larger than exactly $i - 1$ other elements of A.

We can solve the selection problem in $O(n \log n)$ time since we can sort the number using heapsort or mergesort and then simply index the i-th element in the output array.

Part III

Graph Algorithms

Part IV

NP-completeness

Part V

Approximation Algorithms

Part VI