Lecture 22: Query Complexity

Objectives of this lecture

- Generalize comparison model to more general forms of queries.
- Formalize information theoretic bounds. Distinctions between counting inputs and outputs.
- Apply decision tree / information theoretic arguments to more complicated comparisons.

1 Philosophy

The approach by which we showed lower bounds for sorting in the comparison model is quite robust. At the end of the day, the only two facts we used are that each comparison query produced 2 outputs, and that the algorithm may need to produce each of the $n!$ different permutations.

The same approach generalizes to other types of queries where the amount of output per operation is limited. Examples of such queries include comparisons with 3-way answers, or multi-way comparisons. Bounding algorithmic costs by the number of such queries performed is often referred to as query complexity.

We can then lower bound the number of queries needed by taking the number of bits of info each query provided, and divide that against the number of bits need to encode all the outputs that we might need to generate. This is known as the info-theoretic limit, and achieving it leads to some surprisingly tricky (and foundational) algorithmic questions.

2 More Complicated Binary Search

We start by considering a more complicated version of binary search.

Problem 1: Hotter Colder

Given a value n, find some hidden $x \in [1, n]$ using the smallest number of queries of the form of QUERY(y):

- y must be a number in $[1, n]$,

- for every query after the first one, the query returns whether the current one is farther, closer, or same distance, to the previously queried value.

Note that you are not told if $y = x$ if you do get lucky: you must decide exactly what x is by the time you are done.

Note that we dropped the case where x is the same as the guess to simplify the problem. Instead this version restricts us to answer at most once, at the end of all the queries.

This is a much more complicated form of queries than the ones we've looked at so far. The answer to the current query depend not only on the value y that is chosen, but also the answer to the previous query.

However, the same approach for showing a lower bound applies. The answer to each query still has only 3 possibilities, so with k adaptive responses, we can only distinguish 3^k possible hidden values. In other words, we need at least $1 + \log_3 n$ queries to figure out x.

An upper bound of $O(\log n)$ is also not hard to achieve using binary search. Given a mid point y, we can query whether $x \leq y$ or $x \geq y$ by querying $y-1$ and $y+1$ in order: the 'midpoint' of these is y, so anything $< y$ is better for $y-1$, anything $> y$, $y+1$ is better.

This gets $2 \lg n + O(1)$, which is off by a factor of $2 \cdot \lg 3 \approx 2 \cdot 1.58 \approx 3$. It turns out with a bit more work, we can get to $\lg n + O(1)$.

The main idea is to perform binary search, and query $mid * 2 - y$ as the next query point. The difficulty is to ensure that this query point stays in $[1 \dots n]$. For this, we show that as long as the current interval is contained in a valid zone that's $O(1)$ times larger, this drift never causes a problem.

Lemma 1: Query Drift

If we perform binary search on $[l, r]$ by repeatedly querying $2 * midpoint - y^{prev}$, the total distance that y moved throughout the process is $O(r-l)$.

Proof. Repeatedly reflecting a point across different midpoints can move that point (and its reflection) by at most the distnace that the midpoint moves.

The total distance that the query points travel in a binary search is

$$\frac{r-l}{2} + \frac{r-l}{4} + \ldots \leq O(r-l)$$

so we get that the total distance that the reflections move is also bounded as such. □

3 Information Theoretic Lower Bounds Revisited

The proof technique of lower bounding the number of possible outputs, and upper bounding the amount of different answers each query provides, is often called an "information theoretic"

argument. The goal of the algorithm is to gain enough information to distinguish between a number of outcomes (what we refer to as 'output'). If we have some problem with M different outputs that the algorithm needs to be able to produce, and each query returns k different possible answers, then we obtain a worst-case lower bound of $\log_k M$.

Measuring the number of bits for output and outcomes separately essentially stems from the change of base identity for logarithmis

$$\lg_k M = \frac{\lg_M}{\lg_k}.$$

It says that we can estimate the number of binary bits needed to represent the output, as well as the number of binary bits needed to represent each outcome, separately. Specifically, for an algorithm to successfully return something that needs $\lg M$ bits to encode, it must acquire $\lg K$ bits (through the queries) at least $\lg M / \lg k = \log_k M$ times. When reasoning about query counts in this manner, this is often abbreviated to 'output has x bits', or 'we gain y bits per query'.

Going back to the examples:

1. For sorting, we need at least $\lg(n!)$ bits of information about the input before we can correctly decide which output to return.
2. For the probelm of finding some x in $1 \dots n$, we need $\lg n$ bits to encode x

4 Decision Trees Revisited

When optimizing query counts of more complicated queries, the most powerful (but also most computationally expensive) method is still explicitly constructing decision trees. Consider again the hotter-colder problem in Problem 1. It's tempting to think that the 'same' provides so little info that it's basically useless (in the same veins as the 'hit' information).

This turns out to be not the case: there are some values of n where very carefully constructed decision trees do better than the bound of $\lg_2 n$ that one would get should the outputs only be 'hotter' or 'colder'.

The structure of the bound suggest that we should look for better decision trees for n that's just slightly bigger than a power of 2.

In fact, here is a funny strategy that seems to get three queries for $n = 5$:

1. First query $y = 1$
2. Then query $y = 3$:
   (a) If $x = 2$, then we'd get 'same'.
   (b) If $x = 1$, then we'd get 'farther'.
3. So it suffices to consider the case where we get 'closer', which means only $x \in \{3, 4, 5\}$ remain as possibilities.

4. Then query $y = 5$:
   (a) If $x = 4$, then we'd get 'same'.
   (b) If $x = 3$, then we'd get 'farther'.
   (c) If $x = 5$, then we'd get 'closer'

Note that this looks quite different than the binary search strategy, which would start with $y^{(1)} = 2$, $y^{(2)} = 4$. That one is problematic because if answer is on the $1,2$ side, 4 is too far away to 'reflect down'.

For queries as tricky as this, one often need to machine search for good decision trees. From what I remember, the particularly hard cases for this are $n = 17$ and $n = 19$.

5 B-Way Comparisons

This is instead of comparing 2 numbers at a time, we can compare B numbers at a time.

The amount of info provided at each step is $\log_2(B!) \approx B \log B$.

Divide the number of bits of the output space, $n \log n$ by this gives a lower bound of

$$\frac{n \log n}{B \log N} = \frac{n}{B} \frac{\log_n}{\log B} = \frac{n}{B} \log_B n.$$

An upper bound of $(n/B)\log n$ is not hard: we can perform $B/2$ merge sorts in parallel.

With randomization, getting to $\frac{n}{B} \log_B n$ is also not too hard: if we have $0.1B$ pivots for quick sort, we can figure out where $0.9B$ elements sit between the pivots in one B-way comparison. Then each layer of the B-way recursion takes n/B comparisons.

To get a deterministic procedure, we need to perform $\Theta(B)$-way merges. This essentially reconstructs B-trees. I will discuss a limited version of merging $\Omega(B)$ blocks of size B into one sorted list using $O(B)$ of these queries.

The main idea is as follows:

1. Maintain a list of 'dividers' with at most B elements between them.
2. When inserting a new block of say, $B/2$ elements, sort it together with the at most $B/2$ dividers, to figure out how many to distribute into each block.
3. Whenever a block reaches B, sort it completely, break it into two blocks of size $B/2$, and take its median as a new 'divider'.

Then at the end of the day we can sort each block, to get a sorted list.

The number of times we rebuild a block needs to be bounded by amortized anaysis. Each time a block size exceeds B, it's because at least $B/2$ elements are inserted to it. Because only $O(B^2)$ elements are inserted in total, this happens at most $O(B)$ times.

This generalizes to 3 layers and etc. But the constant slack factors above start to compound, so I'll stop here.