CS 325 Recitation Session 3

Chirag Kaudan

May 2025

1 Introduction

- Survey of how many people have grading questions/complaints
- Are people confused about some conceptual stuff from lecture?
- Goal for today: Get familiar with more recursive problem solving and divide and conquer algorithms

2 Tower of Hanoi

Nice puzzle from the mathematician Lucas (known for number theory, Fibonacci sequences, etc.). Start with the 3 peg and 3 disc example.

Basic Setup: We are given a tower of n disks initially stacked in decreasing size on one of 3 pegs. The goal is to transfer the entire tower of disks to one of the other pegs, moving only one disk at a time and never moving a larger disk onto a smaller disk.

Lucas told the ancient legend of the tower; at the beginning of time, God placed 64 golden disks on the first of three diamond needles and told a group of priests that they must transfer every disk to the third needle according to the above rules. These priests work towards this task everyday. When they finish, the tower will crumble and the world will end.

2.1 Solving The Problem Optimally

The first order of business is to think about whether or not the problem even has a solution; should eventually be convinced that we can always do this. The question now becomes, *what is the minimum number of moves in which we can do this*? Playing around with small cases is helpful!

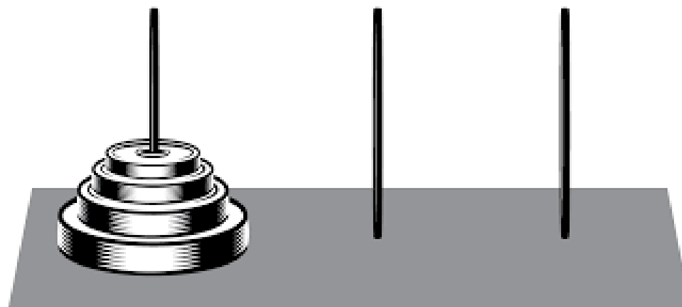


Figure 1: The $n = 4$ case; picture from Illustrative Mathematics Algebra 2

Notation: Let's say that $T(n)$ is the minimum number of moves to transfer n disks from one peg to another peg using the rules. Immediately, we have the lower bound of $T(n) \geq n$ for all $n \geq 1$, since you have to touch each disk at least once to complete the task. But this is not too helpful, and indeed the world may have ended already if it really only took 64 moves to solve the Tower of Hanoi.

We know from experimenting that $T(1) = 1$ and $T(2) = 3$. We could even say that $T(0) = 0$ with the correct interpretation (0 moves to complete the problem if it has 0 disks).

Through experimentation with $n = 3$, we come up with an idea. First move the $n-1$ smallest disks onto the middle peg (taking $T(n-1)$ moves), then move the largest disk to the last peg (1 move), then again move the $n-1$ smallest disks to the last peg (again taking $T(n-1)$ moves) atop the largest disk. This suggests recursive thinking!

2.2 Recurrence

We have formally justified the following recurrence:

$$T(n) \leq 2T(n-1) + 1, \qquad \text{for } n > 0$$

Note that there is an $\leq$ sign in this recurrence, not an $=$ sign. This is because so far we have only proven that you never have to take more than $2T(n-1)+1$ to complete the Tower of Hanoi on n disks. How do we know that there isn't a faster way?

We make the following observation:

Observation: At some point, we must move the largest disk. For this to happen, the smallest $n-1$ disks must be all on a single peg; otherwise, we cannot move the largest disk anywhere and we are stuck. Furthermore, by the rules of the game, if the smallest $n-1$ disks are all on a single peg, then they must be stacked in decreasing size (like they originally were)

This observation forms the core idea for an argument that nobody can solve the tower of Hanoi better than the recursive solution we just gave above. That is, we have proven the following:

$$T(n) \geq 2T(n-1) + 1, \qquad \text{for } n > 0$$

The two inequalities along with the trivial solution of $T(0) = 0$ give the following recurrence

$$\begin{aligned} T(0) &= 0 \\ T(n) &= 2T(n-1) + 1, \qquad n > 0 \end{aligned} \tag{1}$$

2.3 Solving the Recurrence: A Closed Form

With this recurrence, we can compute $T(n)$ for any $n \geq 0$. But we are usually happier when we can also solve the recurrence; obtain a "closed-form solution" to it. There are a few ways to solve a recurrence in general.

2.4 Method 1: Guess and Check

One good way to solve a recurrence is to first guess a solution and prove it is correct. This requires some insight/intuition or pattern recognition so we do what is often helpful: look for any patterns in the first few solutions.

$T(3)$	$2 \cdot 3 + 1 = 7$
$T(4)$	$2 \cdot 7 + 1 = 15$
$T(5)$	$2 \cdot 15 + 1 = 31$
$T(6)$	$2 \cdot 31 + 1 = 63$

A pattern emerges! It looks like $T(n) = 2^n - 1$ for $n \geq 0$. But we have only shown that this is true for $n \leq 6$, we use **mathematical induction** to prove it is true for any $n \geq 0$. The basis step is $T(0) = 0 = 2^0 - 1$. For the inductive step, let $n \geq 1$ and assume that the solution is correct for integer $n-1$. Then $T(n) = 2T(n-1) + 1 = 2(2^{n-1} - 1) + 1 = 2^n - 2 + 1 = 2^n - 1$, which concludes the inductive step. Thus our guess is correct.

So in general, the Tower of Hanoi problems takes a **long** time. For the priests in the Tower of Hanoi legend, it will take $2^{64} - 1$ or 18 quintillion ($18 \cdot 10^{18}$). If they performed one move per second, it would still take over 580 billion years; looks like we are safe for a while!

2.5 Method 2: Algebraic Trickery

How do we solve recurrences if we have no clue how to come up with a good guess? Notice the following, which we get from adding 1 to Equation (1) above:

$$\begin{aligned} T(0) + 1 &= 1 \\ T(n) + 1 &= 2T(n-1) + 2, \qquad \text{for } n > 0 \end{aligned}$$

And if we let $U(n) = T(n) + 1$ then,

$$\begin{aligned} U(0) &= 1 \\ U(n) &= 2U(n-1), \qquad \text{for } n > 0 \end{aligned}$$

Hopefully it is easier to see that the solution to this recurrence is much more manageable; simply $U(n) = 2^n$ and therefore $T(n) = U(n) - 1 = 2^n - 1$.

In general, we will be going through these steps many times when using any type of recursion in algorithms, including in Divide and Conquer algorithms:

- Use recursive thinking to solve a problem; prove your recurrence is correct (sometimes using mathematical induction)
- Come up with a closed form, usually solving the recurrence with mathematical induction

Induction and recurrences play very well with each other!

3 Counting Inversions

Before we get into the problem, here is some context and motivation.

3.1 Applications/Motivation

The problem we consider has to do with rankings. Say that you and a friend rate some movies and want to compare your rankings; what's a good measure of doing this? A lot of websites and systems use something called **collaborative filtering** which are recommender systems where they try to match your preferences of things with other people on the Internet. They do this by comparing your rankings of certain things (like brands or products) to other people's rankings, and then once it has grouped you with people that have similar preferences, it starts recommending stuff to you based off of what other people like. This is a type of machine learning.

But how do you compare two rankings?

> **Making it more concrete:** What's a good way to quantify how similar two rankings are to each other? We want a **measure** that tells us how similar two rankings are.

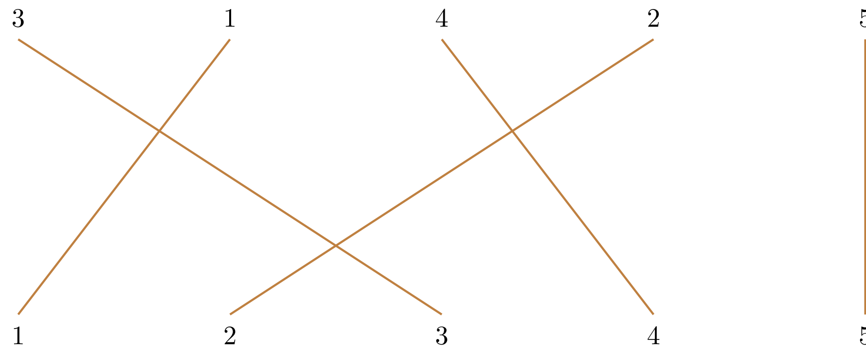
Natural idea: let's say me and a friend were ranking the first 5 letters of the alphabet (people will rank anything these days).

Rank	You	Friend
1	E	B
2	B	C
3	D	E
4	A	A
5	C	D

For each list, check how many of the pairs are "out of order" compared to some fixed ordering. We can make this more concrete.

Definition 1. *Given a sequence $a_1, \dots, a_n$, we say that two indices $i < j$ form an* ***inversion*** *if $a_i > a_j$*

Example: Consider the sequence $3, 1, 4, 2, 5$. It has the following inversions: $(3, 1), (3, 2), (4, 2)$. Geometrically, pair of crossing lines is an inversion



Inversions are nice because the fit the intuition of how to measure how "tangled up" a sequence is compared to another one. Identical sequences have 0 inversions; two completely opposite (reversed) sequences have $\binom{n}{2}$ inversions, which is the most possible.

Problem Statement: Given a set of n distinct numbers $a_1, \dots, a_n$, count how many inversions there are.

3.2 Designing an Algorithm

The easiest, naive algorithm is just to look at each pair (a_i, a_j) of numbers in the sequence and compare them. An algorithm designed around this would take $\Theta(n^2)$ time ($\binom{n}{2} = \frac{n(n-1)}{2} = \Theta(n^2)$), since we would have to look at each inversion individually.

Notice that this problem is very sensitive to moving around numbers; if we ever move around numbers during our algorithm, it can drastically change the number of inversions we have.

Despite this intuition, the algorithm we design will sort numbers, but only after capturing a count of how many inversions are in those numbers. In some sense, we are restricting the mess of moving numbers around in this problem by containing it in a subarray and capturing the crucial information.

Idea (Divide and Conquer): Divide the list in half, count the number of inversions in each half, and then count the number of inversions across the halves i.e. count the number of inversion pairs (a_i, a_j) where a_i is in the first half and a_j is in the second half.

Like mergesort, suppose we have divided the input list into two halves, A and B. We have sorted each of A and B and also have counted the number of inversions in each of the lists. How do we count the inversions

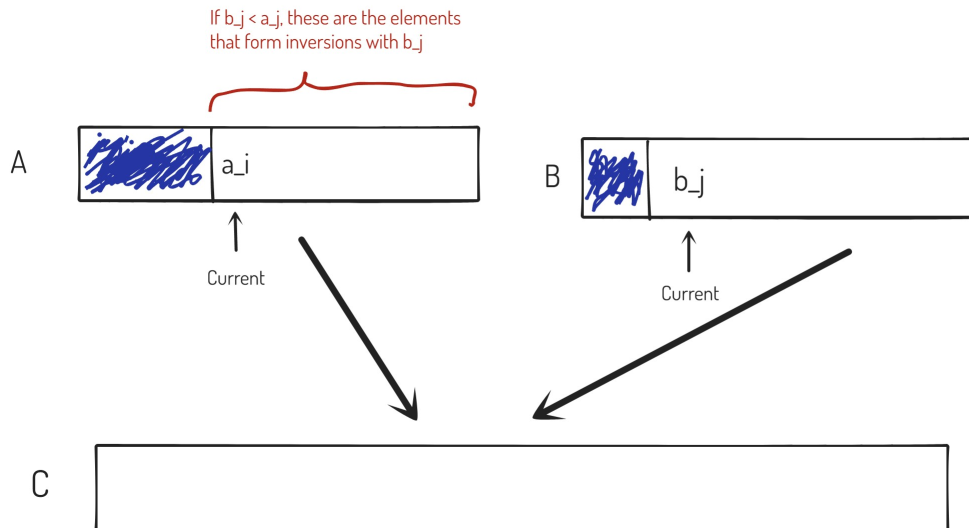


Figure 2: One step of the Merge routine, now accommodating counting inversions. Sorted lists A and B and this is the case where $b_j < a_j$.

that are across A and B? **Key idea:** we use the fact that the lists have already been sorted to easily count inversions.

Again like mergesort, we can do the linear time Merge(A, B) routine (recall from class), which takes in two sorted lists and returns a sorted list C which is the union of lists A and B.

In Merge, we maintained pointer *Current* for each of A and B which pointed to the front of the lists; supposing that *Current* was pointing at a_i in A and b_j in B, we compared a_i and b_j and appended the smaller of the two at the end of list C.

Observation: Each time a_i is appended to list C, we know that no new inversions are encountered because a_i is smaller than b_j, which is smaller than everything else in list B. However, each time that b_j is appended to list C, this means that b_j is smaller than a_i and therefore smaller than everything in list A, so we have several inversions. Specifically, the number of inversions involving b_j would be however many elements are remaining in list A

This lends itself to an algorithm (we are assuming distinct numbers):

Algorithm 1: Merge-and-Count(A, B)

```
   Input : Two sorted lists of numbers A and B
 1 Maintain pointer Current into each list, initially pointing to first elements
 2 Let Count = 0 // This will maintain the number of inversions counted so far
 3
 4 while |A| and |B| ≠ 0 do
 5     if a_i < b_j // Here a_i and b_j are what they mean above; the elements from lists A
           and B resp.  that are being pointed at currently
 6     then
 7         Append a_i to output list
 8         Set Current of list A to next element
 9     if a_i > b_j then
10         Append b_j to output list
11         Increment Count by the number of elements remaining in A
12         Set Current of list B to next element
13 When one list becomes empty, append remainder of other list to the output list
14 return Count and output list
```

Theorem 1. Merge-And-Count(A, B) *runs in time* $\Theta(n)$ *where* n *is the length of the original input sequence (the sum of the lengths of* A *and* B*).*

Proof. Each operation takes constant time work outside the While loop. Inside the loop, we process each element at most once, never dealing with it again, so each iteration of the While loop runs in constant time. Therefore, the number of iterations can be at most the sum of lengths of A and B, which is n. This leads to $O(n)$ running time bound on Merge-And-Count(A, B). It is easy to show that the running time is also $\Omega(n)$ (in the context of asymptotic running time analysis in our class... outside that context, we can explicitly just say that it has a worst case running time of $\Theta(n)$ or really even just saying "it runs in $O(n)$ time is colloquially acceptable). □

We can use the above algorithm as a subroutine in the final algorithm to solve the problem.

Algorithm 2: Sort-and-Count(L)

```
  Input : A sequence/list L of n distinct integers
1 if L has 1 element then
2     return 0 inversions and the list L
3 Divide L into two halves: A containing first ⌈n/2⌉ elements and B containing the remaining ⌊n/2⌋
4 (r_A, A) ← Sort-and-Count(A)
5 (r_B, B) ← Sort-and-Count(B)
6 (r, L) ← Merge-and-Count(A, B)
7 return r = r_A + r_B + r and the sorted list L
```

3.3 Running Time Analysis

This algorithm has the same asymptotic running time as Mergesort. By Theorem 1, we know that the Merge-and-Count subroutine takes $\Theta(n)$ time. So if $T(n)$ is the running time of our algorithm, Sort-and-Count satisfies the following recurrence

$$T(1) \le c$$
$$T(n) = 2T(n/2) + \Theta(n)$$

Which from Mergesort analysis, you should recognize has solution $T(n) = \Theta(n \log n)$.

3.4 Summary

We designed an $\Theta(n \log n)$ algorithm for the counting inversion problem, **substantially** faster than the $\Theta(n^2)$ naive algorithm which looks at every pair of numbers.

In this problem, we saw how the ideas behind the Mergesort algorithm could be generalized to solve a problem that was not directly related to sorting numbers. Multiple divide and conquer style algorithms have similar relationships with each other.

4 Conclusion

We talked about devising recurrences for problems using recursive thinking. We solved those recurrences (via mathematical induction) and talked about manipulating recurrences. We used the Tower of Hanoi to demonstrate these mathematical techniques and the Counting Inversions problem to further illustrate recursive thinking and algorithm design.