CS1570 Lecture Notes

Claire Mathieu

Fall 2026

1 September 15: Introduction to Divide-and-Conquer

The divide-and-conquer paradigm breaks (divide) a problem down into subproblems of the same type that are solved recursively. The solutions to the subproblems are then combined (conquer) to give a solution to the original problem.

Last time we studied greedy algorithms, and the interesting part of the analysis was rthe proof of correctness. Today we study divide-and-conquer algorithms, and for such methods, the interesting part of the analysis typically is the analysis of runtime, so that's what we focus on today.

1.1 Merge Sort

Merge Sort is a classic sorting algorithm that utilizes divide-and-conquer:

- **Divide:** Split the array of size n into two halves of size $n/2$. (Runtime: $O(1)$ or $O(n)$ depending on implementation specifics)
- **Recurse:** Recursively sort both halves. (Runtime: $T(n/2) + T(n/2)$)
- **Conquer:** Merge the two sorted halves into a single sorted array. (Runtime: $O(n)$ with any resonable implementation)

We omit the description of the merge step but any reasonable implementation of it takes linear time. The recurrence relation for Merge Sort is thus:

$$T(n) = 2T\left(\frac{n}{2}\right) + \Theta(n) \tag{1}$$

With the base case $T(1) = 1$, solving this recurrence yields:

$$T(n) = O(n \log n) \tag{2}$$

(Proving this from the recurrence: exercise.)

1.2 Binary Search

Binary Search finds the position of a target value within a sorted array:

- **Divide:** Find the middle element. Check if the target x is greater than or less than the middle element. ($\Theta(1)$ time)
- **Recurse:** Recurse on the relevant half of the array (size $n/2$). (Runtime $T(n/2)$)
- **Conquer:** Deduce the rank of x in the original array, and output that. (Runtime $O(1)$)

The recurrence relation is thus:

$$T(n) = T\left(\frac{n}{2}\right) + O(1) \tag{3}$$

With the base case $T(1) = 1$, solving this yields:

$$T(n) = O(\log_2 n) \tag{4}$$

(Proving this from the recurrence: exercise.)

1.3 Matrix Multiplication

1.3.1 Naive Divide-and-Conquer Approach

Consider multiplying two $n \times n$ matrices, A and B, to produce $C = AB$, using deivide-and-conquer.

- **Divide:** We can partition each matrix into four $\frac{n}{2} \times \frac{n}{2}$ submatrices:

$$A = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix}, \quad B = \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix} \tag{5}$$

  (Runtime $O(n^2)$)

- Recursively compute all the products $A_{ik}B_{kj}$ for $i, j, k \in \{1, 2\}$. (Runtime $8T(n/2)$)

- **Conquer:** Using the results of the recursive calls, do matrix additions to compute the resulting submatrices of C:

$$\begin{aligned} C_{11} &= A_{11}B_{11} + A_{12}B_{21} \\ C_{12} &= A_{11}B_{12} + A_{12}B_{22} \\ C_{21} &= A_{21}B_{11} + A_{22}B_{21} \\ C_{22} &= A_{21}B_{12} + A_{22}B_{22} \end{aligned}$$

  Output

$$C = \begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix}, \tag{6}$$

  (Runtime $O(n^2)$ since matrix addition takes time $O(n^2)$)

The recurrence relation for the runtime is thus

$$T(n) = 8T\left(\frac{n}{2}\right) + O(n^2) \tag{7}$$

with the base case $T(1) = 1$. Expanding this recurrence via substitution:

$$\begin{aligned} T(n) &= cn^2 + 8T\left(\frac{n}{2}\right) \\ &= cn^2 + 8\left(c\left(\frac{n}{2}\right)^2 + 8T\left(\frac{n}{4}\right)\right) = cn^2 + 2cn^2 + 8^2T\left(\frac{n}{2^2}\right) \\ &= cn^2 + 2cn^2 + 2^2cn^2 + 8^3T\left(\frac{n}{2^3}\right) \\ &= \cdots \\ &= cn^2(1 + 2 + 2^2 + \cdots + 2^{i-1}) + 8^iT(\frac{n}{2^i}) \end{aligned}$$

The total work is dominated by the last terms. The recurrence stops when i is such that $n = 2^i$, giving

$$T(n) = O(n^3) \tag{8}$$

1.4 The Master Theorem

The three above recurrences all follow the same pattern: the divide and conquer algorithm does a recursive calls on smaller inputs, of size n/b, plus some additional work $f(n)$. This leads to a fairly general divide-and-conquer recurrence of the form:

$$T(n) = aT\left(\frac{n}{b}\right) + f(n) \quad (\text{where } a \geq 1, b > 1, f(n) > 0) \tag{9}$$

The three standard cases are defined by comparing $f(n)$ to $n^{\log_b a}$:

1. If $f(n) = O(n^{\log_b a - \epsilon})$ for some $\epsilon > 0 \implies T(n) = \Theta(n^{\log_b a})$.
2. If $f(n) = \Theta(n^{\log_b a}) \implies T(n) = \Theta(n^{\log_b a} \log n)$.
3. If $f(n) = \Omega(n^{\log_b a + \epsilon})$ for some $\epsilon > 0$, and satisfies the regularity condition $af(n/b) \leq \gamma f(n)$ for $\gamma < 1 \implies T(n) = \Theta(f(n))$.

Appealing to the master theorem be a convenient way to sidestep the few lines of calculations that are otherwise required, so it is good to be aware of its existence.

The proof of the master theorem is no harder than the proofs of the three special cases.

Note that in my entire career working on algorithms, I have never ever had to worry about the regularity condition.

The three examples were meant as a warmup – either straightforward or well-known. Now we turn to interesting, new content about divide-and-conquer algorithms.

1.5 Strassen's divide-and-conquer Algorithm for matrix multiplication (1969)

Strassen's method reduces the number of submatrix multiplications from 8 to 7 by defining 7 strategic products in the recursive step, after preparing the work by doing a bunch of matrix additions and substractions in the "divide" step:

$$\begin{aligned}
P_1 &= (A_{11} + A_{22})(B_{11} + B_{22}) \\
P_2 &= (A_{21} + A_{22})B_{11} \\
P_3 &= A_{11}(B_{12} - B_{22}) \\
P_4 &= A_{22}(-B_{11} + B_{21}) \\
P_5 &= (A_{11} + A_{12})B_{22} \\
P_6 &= (-A_{11} + A_{21})(B_{11} + B_{12}) \\
P_7 &= (A_{12} - A_{22})(B_{21} + B_{22})
\end{aligned}$$

In the "conquer" step, the submatrices of C can then be reconstructed using only a few matrix additions and subtractions:

$$\begin{aligned}
C_{11} &= P_1 + P_4 - P_5 + P_7 \\
C_{12} &= P_3 + P_5 \\
C_{21} &= P_2 + P_4 \\
C_{22} &= P_1 - P_2 + P_3 + P_6
\end{aligned}$$

The reader can easily check that this is indeed correct. Observe that the difference is that the algorithm does many more matrix additions and subtractions (10 in the "divide" step and 8 in the "conquer" step, instead of just 4 matrix additions total in the naïve algorithm) but only 7 matrix multiplications (instead of 8 in the naïve algorithm). The resulting recurrence relation is thus:

$$T(n) = 7T\left(\frac{n}{2}\right) + O(n^2) \tag{10}$$

By the Master Theorem (see next subsection), this evaluates to:

$$T(n) = \Theta(n^{\log_2 7}) \approx O(n^{2.81}) \tag{11}$$

How did Strassen come up with this, in 1969? By magic.

Note: As of modern research, the optimal exponent has been reduced further towards $O(n^{2.371177})$.

1.5.1 Matrix Product Verification

What if you delegate the computation of matrix multiplication to some complicated mechanism that you don't quite understand, and you just want to make sure that the result is correct? That is the problem of verification: Given three matrices A, B, C, verifying whether $C = A \cdot B$. It turns out that that's much easier and faster than doing the multiplication: it can be performed probabilistically in time $O(n^2)$, with a small error probability. You will learn about it if you take Eli Upfal's course.

1.6 Deterministic Median Finding in Linear Time (incomplete)

Developed by Blum, Floyd, Pratt, Rivest, and Tarjan (1972), this algorithm computes the median of an unsorted array in the worst-case linear time. The legend says that the way in which the algorithm was developped was that those five researchers were at the same conference, they sat at the same table for dinner, and one of them asked whether one could find the median faster than in $O(n \log n)$. A lively dinner conversation ensued, and by the end of dinner, they had a linear-time algorithm. (In other words, even before AI came into the picture, it was possible to find some breakthrough results in a matter of hours, just by putting the right set of people together.)

1.6.1 Algorithm Steps: a rough, not-quite-correct approach

1. **Divide: Group Elements, Find Local Medians:** Divide the n elements into groups of 5. Sort each group individually ($\Theta(1)$ per group, $\Theta(n)$ overall) and extract its median.
2. **Recurse:** Recursively find the median of medians: the median m of the $\frac{n}{5}$ extracted medians. This step takes $T(\frac{n}{5})$ time.
3. **Partition:** Compare every element to m to determine its absolute rank in the full set of n elements ($\Theta(n)$ time).
4. **Recurse:** Regardless of m's rank, it is mathematically guaranteed that at most $\frac{7}{10}n$ elements remain as potential candidates for the target median. Recurse on the remaining candidates.

This is not quite correct for the following example. Imagine m has rank $.6n + 1$. After the first 3 steps you know the $.6n$ elements within which the median of the n elements is hidden somewhere. What is the recursive call in step 4? It is not to find the median of the remaining $.6n$ elements: that would give the wrong result (an element that would have rank $.3n$ among the n elements). No. What you want is the element that, among the remaining $.6n$ elements, has rank $.5n$.

So this example shows that sometimes you need to change the problem in order to get a recursive structure.

In this case, the problem that is amenable to recursion is: given a set of n elements *and a rank* k, find the element of rank k among the n elements.

To be continued.

1.6.2 Recurrence Analysis

Combining the steps of the above (incorrect) algorithm yields the recurrence relation:

$$T(n) = T\left(\frac{n}{5}\right) + T\left(\frac{7}{10}n\right) + O(n) \tag{12}$$

Notice that this cannot be solved by a blind application of the master theorem, because the two recursive calls are on subsets that have different sizes! However, since $\frac{1}{5} + \frac{7}{10} = \frac{9}{10} < 1$, the total work across levels decreases geometrically, leading to:

$$T(n) = O(n) \tag{13}$$

To be continued

1.7 Post-lecture question: Asymptotic Notation Definitions

- **Big-O (O):** Upper bound. $f(n) = O(g(n))$ if $\exists c, n_0 > 0$ such that $\forall n \geq n_0, f(n) \leq c \cdot g(n)$.
- **Big-Omega (Ω):** Lower bound. $f(n) = \Omega(g(n))$ if $\exists c, n_0 > 0$ such that $\forall n \geq n_0, f(n) \geq c \cdot g(n)$.
- **Theta (Θ):** Tight bound. $f(n) = \Theta(g(n))$ if $\exists c_1, c_2, n_0 > 0$ such that $\forall n \geq n_0, c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n)$.