CS1570 Lecture Notes

Claire Mathieu

Fall 2026

0.1 September 17: completing linear-time median-finding algorithm

Last time we developped a not-quite-correct divide-and-conquer algorithm for finding the median of a set of numbers, and, for the runtime, a recurrence which I did not solve.

The problem with the algorithm drafted last time was that median-finding does not easily lend itself to recursion: once we have a candidate m, determine its exact rank and how each other element compares to it, and use that to eliminate all elements that are on the smaller side of m, the remaining problem is to find the median *of the original set* within the remaining elements. But that is not the median of those remaining elements!

For example, if m has rank $.6n + 1$, then we can use use to get rid of all the elements greater than itself (and itself), but then, in the remaining $.6n$ elements, our goals is not the median of those remaining elements, but the element of rank $.5n$.

To get around this issue, the idea is to solve a slightly different problem: given a set of n numbers and an integer k, find the element of rank k (for simplicity we are assuming that all elements are distinct). Here is the pseudocode. (I modified the pseudocode provided by an LLM to get rid of the annoying stuff that is necessary for correctness of code, for example the integer divisions and what if n is not divisible by 5; those cumbersome details hide the main algorithmic ideas; I trust that, should you code this, you would be able to turn it into proper code including all those details.)

Algorithm 1 Deterministic Linear-Time Selection

```
1: procedure SELECT( array A of n numbers, integer k)
2:     if n = 1 then
3:         return A[1]
4:     end if
5:     Divide A into n/5 groups of size 5. Sort each group and find its median. Let M be a new array
       containing the medians of all n/5 groups.
6:     Recursively find the median of M: m ← SELECT(M, |M|/2)
7:     Compare every element of A to m to determine its rank, r.
8:     if k = r then
9:         return x
10:    else if k < r then
11:        return the result of the recursive call SELECT({elements of A smaller than m}, k)
12:    else
13:        return the result of the recursive call SELECT({elements of A greater than m}, k − r)
14:    end if
15: end procedure
```

Take-home message: To use the divide-and-conquer paradigm, sometimes it is necessary to solve a slightly more general problem.

This is akin to proofs by induction: sometimes in order for the induction to work you need to prove a slightly more general statement than what is actually your endgoal.

Now, for the runtime analysis, last time I gave the recurrence but did not prove that it worked. It looks

complicated but unfloding the recurrence for a couple of iterations will reveal a pattern.

$$T(n) = T(n/5) + T(7n/10) + cn$$

$$T(n) = cn + (c\frac{n}{5} + T(\frac{1}{5}\frac{n}{5}) + T(\frac{7}{10}\frac{n}{5})) + (c\frac{7n}{10} + T(\frac{1}{5}\frac{7n}{10}) + T(\frac{7}{10}\frac{7n}{10}))$$

$$T(n) = cn + (c\frac{n}{5} + c\frac{7n}{10}) + c(\frac{1}{5}\frac{n}{5} + \frac{7}{10}\frac{n}{5} + \frac{1}{5}\frac{7n}{10} + \frac{7}{10}\frac{7n}{10}) + \sum T(\text{stuff})$$

$$T(n) = cn + cn\frac{9}{10} + cn(\frac{9}{10})^2 + \sum T(\text{stuff})$$

From that, we conjecture that

$$T(n) \leq \sum_{i \geq 0} cn(\frac{9}{10})^i = 10cn.$$

It only remains to prove by induction, using the recurrence and the base case (which I left unspecified), that $T(n) \leq 10cn$ (left as an **exercise**).

A few caveats. The recurrence is not actually an equality but a $\leq$. That still works. All the numbers are integers, so there is a floor or a ceiling here and there, so we need a little bit of slack to make things work rigorously; I am ignoring those considrations, that are insignificant, but an complete proof would have to handle those.

1 September 17: Introduction to Dynamic Programming

We have seen the greedy paradigm. We have seen the divide-and-conquer paradigm. The third basic algorithmic paradigm is dynamic programming.

Dynamic programming (DP) is a method for solving complex problems by breaking them down into smaller subproblems, solving each subproblem, and using those solutions to deduce a solution to the overall problem. The difference with divide and conquer is that the smaller subproblems can overlap, and there can be many of them – as long as when we unfold the recurrence and look at the subsubproblems, the subsubsubproblems, etc, the total number of problems thus encountered remains bounded by a polynomial. Dynamic programming then uses a table to store the solutions to the subproblems as they are computed, to avoid re-computing the solution to the same subproblem several times.

Thus there are 3 ingredients to the design of a dynamic program:

1. a recurrence relation
2. a table to store the solutions to subproblems
3. an order in which to fill that table so that when we try to use the recurrence, the table entries whose values we need are ready at the time when we need to look them up.

Let us make those three ingredients concrete by doing this work for a very simple problem.

1.1 Example: The Staircase Problem

Problem Statement: How many ways are there to climb a staircase with n stairs, if you can only go up 1 or 2 stairs at a time?

Let $f(n)$ be the total number of ways to reach the n-th stair.

1.1.1 1. Recursive Formulation

To reach the n-th stair, the last step taken must have been either a 1-stair jump from the $(n-1)$-th stair, or a 2-stair jump from the $(n-2)$-th stair. This gives us the following recurrence relation:

$$f(n) = f(n-1) + f(n-2)$$
$$f(1) = 1, \quad f(0) = 1$$

This may ring a bell: it is exactly the recurrence to define Fibonacci numbers.

1.1.2 2. Subproblem Table

To avoid recalculating the same states repeatedly (overlapping subproblems), we define a table to store the solutions:

$$F[0 \dots n]$$

such that $F[i]$ is the number of ways to go up i stairs.

1.1.3 3. Iterative Algorithm (Bottom-Up)

Instead of evaluating recursively, we fill the table in increasing order:

1: $F[0] \leftarrow 1$ and $F[1] \leftarrow 1$
2: For $i = 2$ to n
3: $\quad F[i] \leftarrow F[i-1] + F[i-2]$
4: Return $F[n]$

1.2 Matrix Product Parenthesization

Matrix multiplication is associative, meaning that for three matrices A, B, and C, the product can be computed in multiple ways that yield the identical result:

$$ABC = (AB)C = A(BC)$$

However, the computational cost (number of scalar multiplications) can vary drastically depending on the order of parenthesization.

1.2.1 Cost of a Single Matrix Multiplication

Multiplying a matrix A of dimensions $i \times j$ by a matrix B of dimensions $j \times k$ takes time:

$$O(i \cdot j \cdot k)$$

Consider three consecutive matrices with dimensions defined by the indices i, j, k, l:

$$A_{i \times j}, \quad B_{j \times k}, \quad C_{k \times l}$$

There are two primary ways to fully parenthesize this product:

1. $(AB)C$ Total Cost: $i \cdot j \cdot k + i \cdot k \cdot l$
2. $A(BC)$ Total Cost: $i \cdot j \cdot l + j \cdot k \cdot l$

Numerical Example Let the dimensions be defined by $i = 100, j = 100, k = 1, l = 100$:

- **Cost of $(AB)C$:** $(100 \cdot 100 \cdot 1) + (100 \cdot 1 \cdot 100) = 10,000 + 10,000 = 20,000$ operations
- **Cost of $A(BC)$:** $(100 \cdot 100 \cdot 100) + (100 \cdot 1 \cdot 100) = 1,000,000 + 10,000 = 1,010,000$ operations

As shown, choosing the optimal order reduces the computational effort by over $50\times$.

1.2.2 General Problem Formulation

Input: A sequence of matrices $A_1, A_2, \dots, A_n$, where each matrix A_i has dimensions $x_{i-1} \times x_i$.
Objective: Find the most efficient parenthesization to compute the product $A_1 \cdot A_2 \dots A_n$.

1.3 Developing the Dynamic Programming Solution

1.3.1 Why a Greedy Approach Fails

A tentative greedy strategy might be to find the cheapest adjacent pair $A_i \cdot A_{i+1}$, multiply them, and repeat the process. This approach is sub-optimal and does not guarantee a globally minimal cost sequence. (**Exercise**)

1.3.2 Structural Breakdown: The "Last Multiplication"

Any full parenthesization corresponds to a binary evaluation tree where the root represents the very last multiplication performed and the leaves are the matrices.

For instance, if we know that the final multiplication multiplies $(A_1 \dots A_i)$ by $(A_{i+1} \dots A_n)$, then the total cost decomposes into:

1. The optimal cost to compute the product $A_1 \dots A_i$: $C(A_1 \dots A_i)$
2. The optimal cost to compute the product $A_{i+1} \dots A_n$: $C(A_{i+1} \dots A_n)$
3. The cost of the final matruix multiplication: $x_0 \cdot x_i \cdot x_n$

Thus, if the optimal split point i is known, the cost relation satisfies:

$$C(A_1 \dots A_n) = C(A_1 \dots A_i) + C(A_{i+1} \dots A_n) + x_0 x_i x_n$$

Since the optimal split index i is not known beforehand, we must find the minimum over all valid choices of i:

$$C(A_1 \dots A_n) = \min_{1 \le i \le n-1} [C(A_1 \dots A_i) + C(A_{i+1} \dots A_n) + x_0 x_i x_n]$$

1.3.3 Defining Subproblems Generically

A naive function tracking only the size of a prefix, such as $C(n)$, fails because subproblems do not always start at index 1 (e.g., computing an intermediate block like $A_3 A_4 A_5$). However, looking at the recurrence, we see that the subproblems are always about multiplying some consecutive interval of matrices in the sequence. Therefore, we can define our cost function using both a starting index i and an ending index j. I.e. we let $C(i, j)$ be the minimum cost of computing the matrix product $A_i A_{i+1} \dots A_j$.

Recurrence relation. We are now ready to give the recurrence relation:

- **Base Case ($j = i$):** A single matrix requires zero multiplications.

$$C(i, i) = 0$$

- **General Case ($j > i$):**

$$C(i, j) = \min_{i \le l < j} (C(i, l) + C(l + 1, j) + x_{i-1} x_l x_j)$$

DP Table We allocate a two-dimensional table $C[1 \dots n, 1 \dots n]$ to store calculated costs. Because $i \le j$, only the upper triangular portion of the table is utilized.

Filling Order The value of $C[i, j]$ depends on subproblems $C[i, l]$ and $C[l + 1, j]$, which represent smaller sub-chains. Therefore, the table can be filled **diagonally**, by increasing order of $j - i$:

- First, fill the main diagonal where the chain length is 1 ($C[i, i] = 0$).
- Then fill the next diagonal where length is 2 ($C[i, i + 1]$), and progress outward.
- **Output Target:** The final minimum cost for the entire chain is located at $C[1, n]$.

Now that we have gone through the three steps of design, the reader can probably finish and write the pseudo-code for this dynamic programming algorithm (**Exercise**).