Divide and Conquer

Sam McCauley

March 13, 2025

Welcome Back!

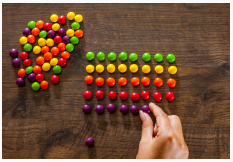
- Problem Set 4 Released
- Uses topics from today. You should be able to get started after today! Will be a little easier after more details on Monday
- I'm almost done grading the midterm; expect it back in the next couple days
  - Grades are pretty good so far!
  - A lot of students were concerned about proofs. A lot of the answers were almost right and got lots of partial credit.
  - I'm not seeing single mistakes "add up" very much
- Problem Set 3 should be back not long after

Divide and Conquer Algorithms

Algorithmic Design Paradigms

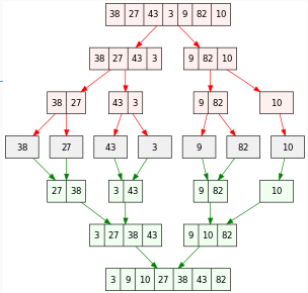
- Greedy Algorithms
  - Gas-filling; maximum interval scheduling
  - Prim's, Kruskal's, Dijkstra's
  - Idea: we choose an item to add *permanently* to the solution
  - Proof that each item we have is correct
- Divide and Conquer ⇐ we are here!
  - Divide problem into multiple parts
  - *Combine* solutions into a new correct solution
- Dynamic Programming
- Network Flow

Sorting



- Selection sort: take largest item; place it in last slot; repeat
- Can be viewed as “greedy:” once we place an item, we have proven that it stays there irrevocably
- $\Theta(n^2)$ time (due to $\Theta(i)$ time to find largest of i items)
- Can we do better with divide and conquer?
- Let’s revisit Merge Sort, and talk about how to analyze it

Merge Sort [von Neumann 1945]



Goal: sort an array A of size n
(Assume $|A|$ is a power of 2 for simplicity)

- If $|A| \leq 1$ then return A
- Otherwise, sort the left half of A and the right half of A using Merge Sort
- "Merge" the two halves together to create a sorted array

Let's look at how to merge efficiently [Blackboard] . Can prove that merge is correct by induction.

Running time of a merge? $O(n)$

Merge Sort

```
1  MergeSort(A, n):
2      if n ≤ 1: return
3      A_1 = A[1, ..., n/2]              % assume integer division
4      A_2 = A[n/2 + 1, ..., n]
5      MergeSort(A_1, n/2)
6      MergeSort(A_2, n − n/2)
7      A = Merge(A_1, A_2)
```

- Let's do a simple example [Blackboard]
- How can we prove correctness?
- *Strong* induction (why?)

Proving Merge Sort Correctness

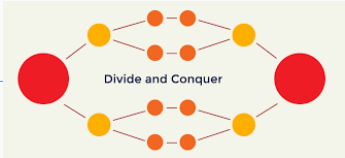
(This slide is for reference)

Inductive Hypothesis: Merge sort correctly sorts any array of size n.

Base Case: if $n = 1$ array is already sorted

Inductive Step (strong induction): If $n > 1$ then we call `MergeSort()` on two arrays; one of size $\lfloor n/2 \rfloor$ and one of size $n - \lfloor n/2 \rfloor$. Since $n > 1$, $\lfloor n/2 \rfloor < n$ and $n - \lfloor n/2 \rfloor < n$, so by *strong induction* these recursive calls sort A_1 and A_2. Since A_1 and A_2 are sorted, `Merge`(A_1, A_2) returns A in sorted order.

Divide and Conquer Running Time



- Analyzing D & C algorithms' running time can be initially confusing
- Challenge: the algorithm "jumps" all over the place due to the recursive structure
- Today: *group/categorize* costs to allow us to analyze divide and conquer more effectively

Merge Sort Running Time

What is the running time of Merge Sort on an array of size n?

One answer:

- running time of Merge Sort on an array of size $n/2$, plus
- running time of Merge Sort on a second array of size $n/2$, plus
- $O(n)$ to merge.
- Or, if $n = 1$, then the cost is 1.

Let $T(n)$ be the *exact* number of operations of Merge Sort on an array of size n. Then:

$$T(n) = 2 \cdot T(n/2) + O(n), \qquad T(1) = 1$$

Recurrences

Recurrences

- To find the running time of a divide and conquer algorithm, we write a *recurrence*
  - A way of specifying the value of a function using the value of that function at smaller points
- Let $T(n)$ be the cost of the algorithm on a problem of size n. Can write $T(n)$ as:
  - A base case for small n (oftentimes $T(1) = 1$)
  - A sum of the "divide" recursive calls which can be written in terms of T (e.g. $T(n/2)$), plus the cost to "conquer"
- A solution to this recurrence gives our total running time!

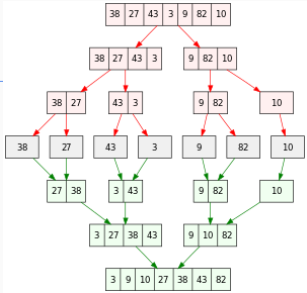
Rounding in Recurrences

- We don't actually split into two arrays of size $n/2$; we actually split into one array of size $\lfloor n/2 \rfloor$ and one of size $n - \lfloor n/2 \rfloor$
- So the recurrence is

$$T(n) = T(\lfloor n/2 \rfloor) + T(n - \lfloor n/2 \rfloor) + O(n).$$

- In this class, you may assume that rounding does not affect the final answer
- In all cases, just write (something like) $T(n) = 2T(n/2) + O(n)$ or $T(n) = T(\sqrt{n}) + O(1)$; assume that all quantities are integers
- The textbook goes into detail about why this is OK

First example: merge sort



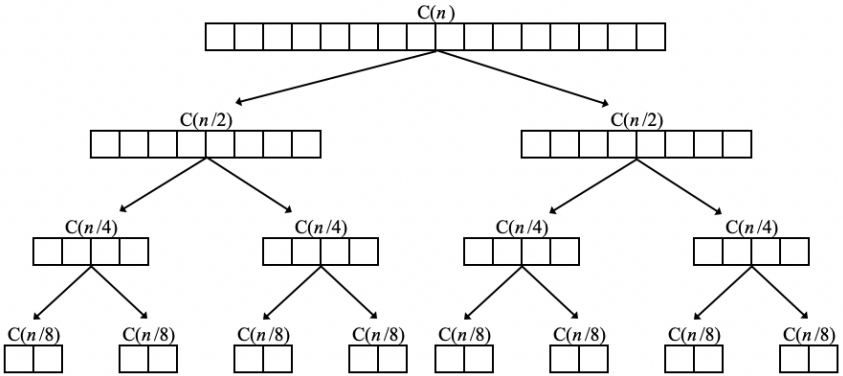
- $T(n) = 2T(n/2) + O(n)$; $T(1) = 1$
- First: set constants
- For some c, $T(n) \leq 2T(n/2) + cn$; $T(1) \leq c$
- How can we solve this?

Recurrence Tree Technique

- Let's draw the recurrence as a tree [Blackboard]
- Idea: this drawing will help us group together the costs of the algorithm
- In what order does Merge Sort actually incur these costs?
- But: can we bound the cost of a given level of the tree?
  - Yes: each level costs cn in total
  - Specifically: level i has 2^i subproblems, each with cost $\leq cn/2^i$
- How many levels are there?
  - Assuming n is a power of 2: $\log_2 n + 1$ levels
- What is the total cost of Merge Sort?

Recurrence Tree Analysis: Merge Sort

- What is this level-by-level analysis saying about Merge Sort?
- Look at *all* work we do across all subproblems of size $n/2^i$
- Answer: cn total work
- So we do cn total work on the subproblem of size n; cn total work on the 2 subproblems of size $n/2$; cn on the four subproblems of size $n/4$, . . ., n on the n subproblems of size 1
- That's $\leq cn(\log_2 n + 1)$ total work!



$\log_2 n$ levels

Merging Costs
n
$2(n/2) = n$
$4(n/4) = n$
$8(n/8) = n$

Total = $n \log_2 n$

Recurrence Tree Method

- Each time the function is called, draw a node in the tree
- (Then, draw a node for each of its children, and so on.)
- What is the *size* of each call?
- What is the *cost* of each call? (The total work done in addition to the recursive calls.) If the cost is big-O, use an explicit constant c
- The total cost of all calls in the tree gives the total value of the recurrence!
- Oftentimes: want to sum row by row
- Probably the most flexible and useful method of solving a recurrence

Double-Checking our Work

- We wanted a solution to:

$$T(n) = 2 \cdot T(n/2) + cn, \qquad T(1) = c$$

- Does $cn(\log_2 n + 1)$ satisfy this?
  - Yes.

$$\begin{aligned} cn(\log_2 n + 1) &= 2\left(\frac{cn}{2}\left(\log_2 \frac{n}{2} + 1\right)\right) + cn \\ &= cn\left(\log_2 \frac{n}{2} + 1\right) + cn \\ &= cn\left(\log_2 n - \log_2 2 + 1\right) + cn \\ &= cn\left(\log_2 n\right) + cn \checkmark \end{aligned}$$

Stepping Back

- Merge Sort divides the array into halves, sorts each half, and then recombines them in $O(n)$ time
- Running time is initially difficult to see
- We wrote the running time as a recurrence
- To solve the recurrence, we drew a tree, which helped us group the costs
- $\log_2 n$ levels, each of cost $O(n)$, means $O(n \log n)$ total cost!

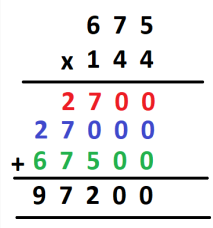
Sorting Algorithm Comparison (Just for Fun)

- Insertion sort is $O(n^2)$, with good constants. Usually best for arrays of $\leq\approx 64$ elements
- Merge sort is $O(n \log n)$; used in Java and Python libraries
  - An optimized version switches to Insertion sort when recursing on at most 64 elements
- Heapsort (sorting using repeated `ExtractMin` from a binary heap), Quicksort (we'll see later in the class) are also fast but less used

Merge Sort

- Classic divide and conquer algorithm; need:
  - A base case
  - A way to divide into smaller instances
  - A way to combine the solution for smaller instances into an overall solution
- What do we need for correctness?
  - Combining smaller solutions must give correct solution for overall instance
  - Base case must be correct
  - Must *reach* the base case!

Divide and Conquer: Multiplication



- Let's say we want to multiply two n-digit numbers $a \times b$ (assume they're in base 10; can extend to binary numbers)
  - Let's say n is too big for our CPU: $n \gg 64$
- What is the running time of the algorithm you learned in school?
  - For each digit of b, multiply with each digit of a; carry as necessary
  - $O(n)$ time for each digit of b
  - $O(n^2)$ time overall
- Addition is only $O(n)$ however
- Can we do multiplication more efficiently? In 1960, Kolmogorov *conjectured* no: any algorithm takes $\Omega(n^2)$ worst-case time

Divide and Conquer: Multiplication

Assume n is a power of 2 for the moment for simplicity.

- Let's write a as the sum of two $n/2$-bit numbers: $a = 10^{n/2}a_\ell + a_r$
  - E.g.: $123456 = 123000 + 456$
- Let's write b as the sum of two $n/2$-bit numbers: $b = 10^{n/2}b_\ell + b_r$
- Then $a \times b = (10^{n/2}a_\ell + a_r)(10^{n/2}b_\ell + b_r)$
- Using algebra, $a \times b = 10^n(a_\ell b_\ell) + 10^{n/2}(a_\ell b_r + b_\ell a_r) + a_r b_r$.

Divide and Conquer: Multiplication

$$a \times b = 10^n(a_\ell b_\ell) + 10^{n/2}(a_\ell b_r + b_\ell a_r) + a_r b_r$$

- So we can use divide and conquer! To multiply two n-digit numbers, we first perform four recursive multiplications:
  - $a_\ell b_\ell$, $a_\ell b_r$, $b_\ell a_r$, and $a_r b_r$
- Can multiply by 10^n in $O(n)$ time by adding 0
- Then we add them together in $O(n)$ time.
- If $n = 1$ just multiply the numbers
- Recurrence?
- $T(n) = 4T(n/2) + O(n)$; $T(1) = 1$
- Let's solve this recurrence together [Blackboard]
- Get $\Theta(n^2)$ time, same as before. *Can we improve this?*

Geometric Sums

This sum is extremely useful when analyzing recurrences.

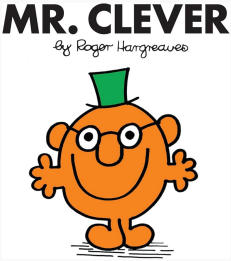
Lemma

For any $r \neq 1$ and integer $k \geq 0$,

$$\sum_{i=0}^{k} r^i = \frac{r^{k+1} - 1}{r - 1}$$

I will add this sum to the notes on logs etc. on the website.

Divide and Conquer: Karatsuba's Algorithm



$$a \times b = 10^n(a_\ell b_\ell) + 10^{n/2}(a_\ell b_r + b_\ell a_r) + a_r b_r$$

- Consider the following *three* recursive multiplications
  - $(a_\ell b_\ell)$, $(a_r \times b_r)$, and $(a_\ell + a_r) \times (b_\ell + b_r)$
- I claim this is enough! Why?
- $a_\ell b_r + b_\ell a_r = (a_\ell + a_r)(b_\ell + b_r) - a_\ell b_\ell - a_r b_r$
- So after *three* recursive calls of size $n/2$ I can calculate $a \times b$. I used $O(n)$ total time other than the recursive calls
- $T(n) = 3T(n/2) + O(n)$; $T(1) = 1$

Solving the Multiplication Recurrence

$$T(n) = 3T(n/2) + O(n) \qquad T(1) = 1$$

- Let's solve this recurrence [Blackboard]
- We want to ask ourselves: What is the height of the tree? What is the cost of each level?
- Solution: $O(n^{\log_2 3}) = O(n^{1.58})$ time
- Much better than n^2!
- Reflect: why did changing a *constant* from 3 to 4 have such an impact on the running time?

Multiplying Numbers Efficiently

- Kolmogorov conjectured that $\Omega(n^2)$ time is needed; stated this conjecture in a seminar at Moscow State University in 1960
- Karatsuba, a student figured out this $O(n^{\log_2 3})$ time algorithm in the next week
- Kolmogorov cancelled the whole seminar and then published the result on Karatsuba's behalf without telling him

Multiplying Numbers Efficiently

- Kolmogorov conjectured that $\Omega(n^2)$ time is needed; stated this conjecture in a seminar at Moscow State University in 1960
- Karatsuba, a student figured out this $O(n^{\log_2 3})$ time algorithm in the next week
- Kolmogorov cancelled the whole seminar and then published the result on Karatsuba's behalf without telling him

Multiplying Numbers Efficiently

- Can we do better?
- Best known: $O(n \log n)$ [Harvey, van der Hoeven 2019]
- Are these speedups useful in practice?
    - Sometimes! Karatsuba's is used in some libraries

Solving Divide and Conquer Problems

You need:

- A way to split the problem into smaller subproblems: smaller instances of the problem we're trying to solve
- Each of these subproblems will be solved recursively: using the same algorithm we are using now
- Then, need to combine the solutions to these subproblems into a solution for our overall problem
- Use strong induction to prove correctness; recurrence to analyze performance