Divide and Conquer

Sam McCauley

March 7, 2024

Welcome Back!

- Assignment released
    - Group assignment
    - main D&C practice so be sure you participate
    - *Can* solve any problems after today; we'll get more practice Monday so it will be easier after that

- Midterm handed back; discussion next slide

Midterm

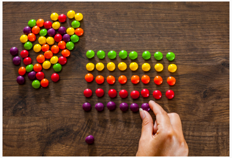
- Back on gradescope
- Median and mean ≈ 87; standard deviation 8
- Final grades are usually a little higher due to assignments
- Midterms are short; having 2 (plus a final) helps things average out in the long run
- Let's look very quickly at a few common sticking points; I'm always happy to have a longer conversation in office hours

Divide and Conquer Algorithms

Algorithmic Design Paradigms

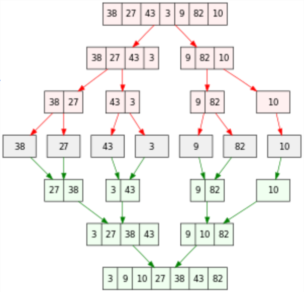
- Greedy Algorithms
    - Gas-filling; maximum interval scheduling
    - Prim's, Kruskal's, Dijkstra's
    - Idea: we choose an item to add *permanently* to the solution
    - Proof that each item we have is correct
- Divide and Conquer $\Leftarrow$ we are here!
    - Divide problem into multiple parts
    - *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 (requires $\Omega(i)$ 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 [On Board #1]. Can we prove that the merge is correct by induction?

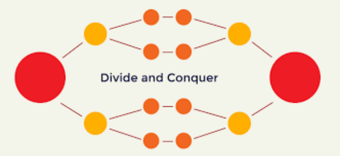
Running time of a merge? $O(n)$

Merge Sort

```
MergeSort(A, n):
      A_1 = A[1, ..., n/2]
      A_2 = A[n/2 + 1, ..., n]
      MergeSort(A_1,n/2)
      MergeSort(A_2,n/2)
      A = Merge(A_1, A_2)
```

- Let's do a simple example [On Board #2]
- How can we prove correctness?
- *Strong* induction (why?)

Divide and Conquer Running Time



- Analyzing D & C algorithms 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:

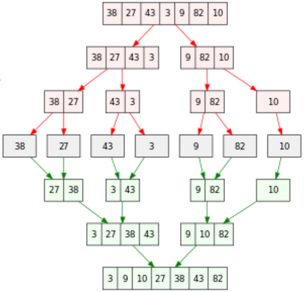
$$T(n) = 2 \cdot T(n/2) + O(n), \qquad T(1) = 1$$

Recurrences

Recurrences

- To find the running time of a divide and conquer algorithm, we write a *recurrence*
- 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!

First example: merge sort



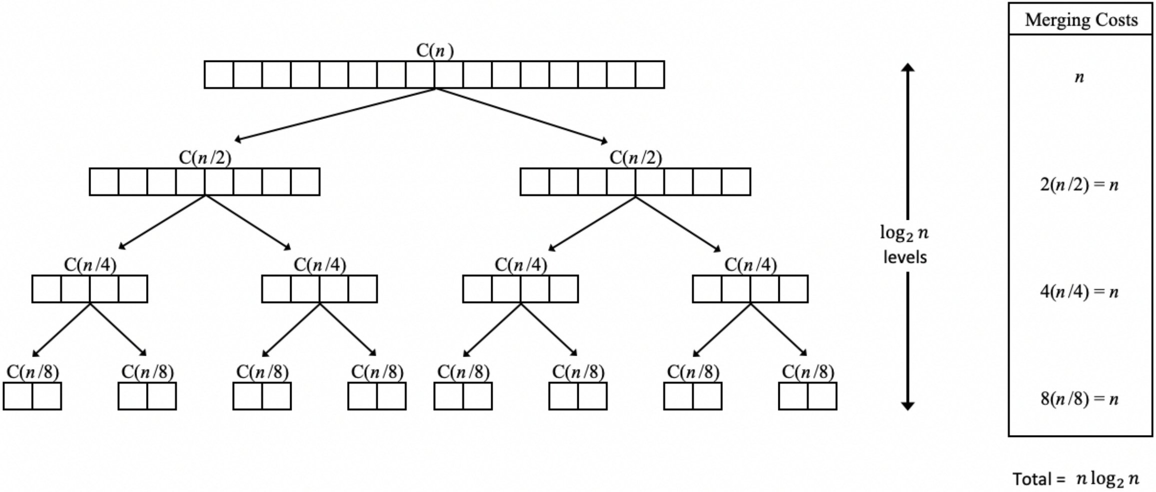
- $T(n) = 2T(n/2) + O(n)$; $T(1) = 1$
- First: set constants
- For some c, $T(n) \leq 2T(n/2) + cn$; $T(1) \leq c$
- How can we solve this?

Recurrence Tree Technique

- Let's draw the recurrence as a tree [On Board #3]
- Idea: this drawing will help us group together the costs of the algorithm
- How does Merge Sort actually run?
- 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?
- 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 $n/4, \ldots, n$ on the n subproblems of size 1
- That's $\leq cn(\log_2 n + 1)$ total work!


C(n)
C(n/2)
C(n/2)
C(n/4)
C(n/4)
C(n/4)
C(n/4)
C(n/8)
C(n/8)
C(n/8)
C(n/8)
C(n/8)
C(n/8)
C(n/8)
C(n/8)
log2 n
levels
Merging Costs
n
2(n/2) = n
4(n/4) = n
8(n/8) = n
Total = n log2 n

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.

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

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 in a bit) 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

$$\begin{array}{r} 675 \\ \times\ 144 \\ \hline 2700 \\ 27000 \\ +\ 67500 \\ \hline 97200 \\ \hline \end{array}$$

- 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:
  - $a_\ell \times b_\ell$, $a_\ell \times b_r$, $b_\ell \times a_r$, and $a_r \times b_r$
- And 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 [On Board #4]
- Get $\Theta(n^2)$ time, same as before. *Can we improve this?*