Divide and Conquer

Divide and Conquer algorithms work by recursively breaking the problem into smaller pieces and solving the subproblems. Then, they combine the results of the sub-problems together.

Examples:

- Mergesort / Quicksort
- Integer Multiplication (Karatsuba algorithm)
- Matrix Multiplication (Strassen's Algorithm)
- Fast Fourier Transform

General Tips:

- Ask Yourself: If I had the answer to a subproblem, how could I get the answer to the whole problem? If this question was answered quite easily, then divide and conquer could give a very nice solution.
- The mergesort recurrence appears quite often. If the run-time is $O(n \log n)$, think about trying to break into 2 pieces and combining them together in linear time.

Practice Problems

Exercise. *Given an array of n elements, you want to determine whether there exists a majority element (that is an element which occurs at least $\lceil \frac{n+1}{2} \rceil$ times) and if so, output this element. Show how to do this in time $O(n \log n)$ using Divide and Conquer. By the way, can you do better?*

Solution.
For $O(n \log n)$, the following straightforward algorithm works. Run recursively on the two halves. If both return the same majority element, then return that element. If only one returns a majority element, or if they both return different elements, run through the combined array in $O(n)$ time to check if one is a majority element.

There exists a linear time solution. Here is one solution: In the first pass of the list, we will find the best candidate for a majority element. In a second pass of the list, we will count the number of occurrences and determine whether the candidate is actually a majority element.

In the first pass, maintain a counter and a reference to the current candidate. At each element in the list, if the counter is 0, then we take the current element as our current candidate and set increment our counter to 1. If the counter is non-zero, then we increment the counter if the current element is the same as the

current candidate and we decrement otherwise. At the end of the pass, if the list has a majority element, then we must have a reference to it.

Here's another solution. Let us go through the array, and every time we see a value, we increment a counter for that value. At the end, we find the value with the largest counter. We check if that counter is at least $\lceil \frac{n+1}{2} \rceil$, and if so, output the corresponding value. What data structure have we encountered that allows us to quickly update a collection of numbers and find the maximum value? A max-heap of course!

This algorithm first initializes an empty heap. As it goes through the array, it inserts the value it sees into the heap if the value is not in the heap already, or increments the counter associated with the value. At the end it calls find_max to identify the value with the largest counter. The running time is $O(n * \text{time of insert/increase-key} + \text{time of find\_max})$. Using a binary heap, this is $O(n \log n)$, and using a Fibonacci heap, this is $O(n)$.

Exercise. *You are given a list of stock prices S of length n where $S[i]$ represents the price of a particular stock on day i. You would like to know what is the most amount of money you could make per share throughout these n days. More formally, compute the maximum of $S[k] - S[j]$ for $1 \leq j \leq k \leq n$, which represents buying on the jth day and selling on the kth.*

Solution.
Break into 2 subarrays of equal size, S_1 and S_2. Either $j, k \in S_1$, $j, k \in S_2$ or $j \in S_1, k \in S_2$. For the first two conditions, compute recursively the maximal profit per share in S_1 and S_2. For the third condition, find the minimum element in S_1 and maximum element in S_2 and take their difference. The final answer for S is the maximum of these three numbers. If we find the min element of S_1 and max element of S_2 by scanning through the arrays, the running time $T(n)$ satisfies the recurrence $T(n) = 2T(n/2) + O(n)$ and so $T(n) = O(n \log n)$. We note that this is quite inefficient though: if we return the minimum and maximum price along with the maximum profit at each step, then we can compute the next maximum profit, maximum price, and minimum price in constant time, giving us $T(n) = 2T(n/2) + O(1)$ and $T(n) = O(n)$.

Here is another linear time solution to this problem. The idea is, we want to buy low and sell high, so let us remember the lowest price we have seen. Initialize current_min to $S[0]$, and max_profit to 0. We iterate through S. For each value v that we see, we set current_min $= \min(\text{current\_min}, v)$ and max_profit $= \max(\text{max\_profit}, v - \text{current\_min})$.

Exercise. *Given an array of distinct integers $A[0 \ldots n-1]$, we say that the ordered pair $(A[i], A[j])$ is an inversion if $i < j$ but $A[i] > A[j]$. Show how to count the number of inversions in an array of size n in $O(n \log n)$ time.*
(Hint: The number of inversions in an array represents how far we are from a sorted array. Think about how we might be able to modify a particular sorting algorithm to count the number of inversions)

Solution.
Using the divide and conquer style of thinking, suppose we were already given the number of inversions in $A[0 : n/2]$ and $A[n/2 : n]$. Then, we would only need to add those two numbers up, and count the number of inversions where $i < n/2$ and $j \geq n/2$, which are the inversions that cross the center of the array.

This doesn't sound like an easy task because if the first half and second half of the array were in any order, then I would need to take every element of my first half and compare it to every element in my second half. However, if the first and second half were sorted, then our task would be a lot easier. This motivates the idea that in addition to counting the number of inversions at each step, we also sort the list to make future counting inversions easier.

This leads to a variation of merge-sort as follows: When I am merging the two sorted half lists H_1 and H_2 into H, whenever I take an element from H_2, I will count the number of elements left in H_1. The remaining elements in H_1 are smaller than this element from H_2, yet we know for sure that all these elements of H_1 came before this element in H_2 in the original ordering. All those elements left in H_1 started out at a smaller index than this removed H_2 element, but have value higher than the removed element. Therefore, we get 1 inversion for each element left in H_1 after taking an element from H_2.

Therefore, our algorithm goes as follows: Run merge sort on the array, but keep track of the number of inversions using a global variable that will be incremented by some number whenever two sub-arrays are merged together.

Exercise. *Consider the pattern matching question from lecture. Given two binary strings s_1 and s_2 of length m and n respectively with $m < n$, we learned how to determine all occurrences of s_1 in s_2.*

We will modify the problem as follows:

(1) Instead of using binary strings, we will consider strings using the digits $\{0, 1, 2, 3\}$.

(2) We define for $a, b \in \{0, 1, 2, 3\}$: $match(a, b) = 1$ if $a = b$; 0.5 if $|a - b| = 1$; and 0 otherwise.

Now, instead of wanting complete matches, we say that $a_1a_2a_3 \dots a_k$ partially matches $b_1b_2b_3 \dots b_k$ if $\sum_{i=1}^{k} match(a_i, b_i) \geq k/2$.

Use FFT to determine the number of partial matches of s_1 in s_2 in $O(n \log n)$ time.

Solution.
Encode the numbers as follows:

$$0{-}11000$$
$$1{-}01100$$
$$2{-}00110$$
$$3{-}00011$$

Now we can easily check that the dot product between two adjacent numbers is 1 and two of the same number is 2. Therefore, we can use exactly what we did in lecture and find all the coefficients of the resulting polynomial that are greater than k (since we get 2 for a complete match and 1 for a half match, everything is scaled by a factor of 2).

Exercise. *Consider an algorithm for integer multiplication of two n-digit numbers where each number is split into three parts, each with $n/3$ digits.*

(a) Design and explain such an algorithm, similar to Karatsuba's algorithm from the first lecture. Your algorithm should describe how to multiply the two integers using only six multiplications on the smaller parts (instead of the straightforward nine).

(b) Determine the asymptotic running time of your algorithm. Would you rather split it into two parts or three parts?

(c) Suppose you could use only five multiplications instead of six. Then determine the asymptotic running time of such an algorithm. In this case, would you rather split it into two parts or three parts?

Solution. (a) Let us consider multiplying a and b, two n-digit numbers. The algorithm will rely on recursion, dividing the n-digit numbers into blocks of $\frac{n}{3}$ consecutive digits, and instead multiplying combinations of these smaller numbers. If n is not divisible by 3, some of these blocks will involve either $\lceil \frac{n}{3} \rceil$ or $\lfloor \frac{n}{3} \rfloor$ digits, but these differences will asymptotically lead to the same running time.

The algorithm's base case for the recursion will be 8 digit by 8 digit multiplication or less, which is very fast on computers. For higher n, consider writing x, y, z as the first, second, and third sections of number a, and p, q, r as the sections of number b. In other words, we have:

$$a = 10^{\frac{2n}{3}} x + 10^{\frac{n}{3}} y + z$$

$$b = 10^{\frac{2n}{3}} p + 10^{\frac{n}{3}} q + r$$

Then we have

$$ab = 10^{\frac{4n}{3}} px + 10^{n}(xq + yp) + 10^{\frac{2n}{3}}(xr + yq + pz) + 10^{\frac{n}{3}}(yr + qz) + rz.$$

Thus, to find ab, we wish to compute px, $xq + yp$, $xr + yq + pz$, $yr + qz$, and rz. We present a method to compute these 5 values with 6 multiplications of two $\frac{n}{3}$ length numbers.

Consider the following products: $xp, rz, yq, (y + z)(r + q), (x + y)(q + p), (x + y + z)(p + q + r)$. We claim these 6 products find the five desired values above. Note that the quantities xp and rz are directly calculated. Furthermore, we have that $(y + z)(r + q) - rz - yq = yr + zq$ and $(x + y)(q + p) - xp - yq = xq + yp$. Finally, now that we have calculated xp, rz, $xq + yp$, and $yr + qz$, we have that $(x + y + z)(p + q + r) - xp - rz - (xq + yp) - (yr + qz) = xr + yq + pz$, giving the final desired quantity.

(b) The runtime for this algorithm is governed by the recursion $T(n) = 6T(\frac{n}{3}) + O(n)$. The additional steps other than recursion during each multiplication are simply linear additions between terms (with some constants). Using the Master Theorem, we have that $T(n) = O(n^{\log_3 6}) = O(n^{1.63\cdots})$. By dividing by 2, we had that $T(n) = O(n^{1.59\cdots})$. Thus, it would in this case be preferable to divide numbers into two parts.

(c) By doing 5 multiplications, we would reduce the runtime to $T(n) = O(n^{log_3 5}) = O(n^{1.46\cdots})$, which is better than the runtime achieved by dividing into two parts. Thus, in this case, it would be preferable to divide the numbers into two parts