

Divide and Conquer: Merge Sort + Solving Recurrences¹

In this and a few coming lectures, we look at the *divide-and-conquer* paradigm for algorithm design. This is applicable when any instance of a problem can be broken into smaller instances, such that the solutions of the smaller instances can be combined to get solution to the original instance. One usually has a “naive” but “slow” method to solve the problem at hand, and the D&C methodology (usually) “wins” if the combining can be done faster than the naive running time.

1 Merge Sort

SORTING AN ARRAY

Input: An array $A[1 : n]$ of integers.

Output: A sorted (non-decreasing) permutation $B[1 : n]$ of $A[1 : n]$.

Size: The number n of entries in A .

Remark: *It is fair to ask why the size of the problem above doesn't include the number of bits required to encode $A[j]$'s and x . The short answer is that it is a modeling choice. If the numbers $A[j]$'s are “small”, that is, they are at most some polynomial in n and thus fit in $O(\log n)$ sized registers, then we assume that simple operations such as adding, multiplying, dividing, comparing, reading, writing take $\Theta(1)$ time. The reason being that for numbers that fit in word/registers indeed these operations are fast compared to the other operations of the algorithm.*

The “naive” algorithm for sorting is one which repeatedly takes the minimum element and puts it in the front; this algorithm takes $O(n^2)$ time. We now show how to do better using divide-and-conquer. This algorithm is *merge-sort*. You have perhaps seen this algorithm before (in CS 10 or CS 1), and the idea nicely captures the Divide-and-Conquer strategy.

First we notice if $n = 1$, then we return the same array; this is the base case. Next, given an input $A[1 : n]$ which needs to be sorted, we wish to divide into two smaller subproblems. We start with a natural way: we divide $A[1 : n]$ into two halves $A[1 : n/2]$ and $A[n/2 + 1 : n]$. Next, we recursively apply the same algorithm to these halves to obtain sorted versions B_1 and B_2 . Note that the final answer that we need is a sorted version of $B_1 \cup B_2$. So in the *combine/conquer* step we do exactly this.

At this point we need to figure out a “win”: why is sorting $B_1 \cup B_2$ any easier than sorting $A[1 : n]$ to begin with. The fact we exploit is that these B_1 and B_2 are individually sorted. We can use this to sort $B_1 \cup B_2$ way faster than the $\Theta(n^2)$ naive algorithm for $A[1 : n]$. This is the non-trivial part of the algorithm, and once we get a “win” here over the naive algorithm, we will see that we get a win over all.

Combine Step. Let us then recall the Combine procedure of MERGESORT

¹Lecture notes by Deeparnab Chakrabarty. Last modified : 19th Mar, 2022
These have not gone through scrutiny and may contain errors. If you find any, or have any other comments, please email me at deeparnab@dartmouth.edu. Highly appreciated!

COMBINE

Input: Two *sorted* arrays $P[1 : p]$ and $Q[1 : q]$.

Output: Sorted array $R[1 : r]$ of elements of $P \cup Q$, with $r = p + q$.

Size: $p + q$.

This is an iterative algorithm which keeps three pointers i, j, k all set to 1. At each step we compare $P[i]$ and $Q[j]$, and $R[k]$ is set to whichever is smaller. That particular pointer and k are incremented. The process stops when either i reaches $p + 1$ or j reaches $q + 1$, in which case the rest of the other array is appended to R . This takes $O(p + q)$ time as each step takes $O(1)$ time, in each step either i or j increments, and so the algorithm terminates in $(p + q)$ steps. A formal pseudocode is given below both of the above combine step and the merge sort.

```
1: procedure COMBINE( $P[1 : p], Q[1 : q]$ ):
2:    $\triangleright$   $P$  and  $Q$  are sorted; outputs  $R$  a sorted array of elements in  $P$  and  $Q$ .
3:    $i = j = k \leftarrow 1$ .
4:   while  $i < p + 1$  and  $j < q + 1$  do:
5:     if ( $P[i] \leq Q[j]$ ) then:
6:        $R[k] \leftarrow P[i]$ 
7:        $i \leftarrow i + 1$ 
8:     else:
9:        $R[k] \leftarrow Q[j]$ 
10:       $j \leftarrow j + 1$ 
11:       $k \leftarrow k + 1$ 
12:   if  $i > p$  then:
13:     Append rest  $Q[j : q]$  to  $R$ 
14:   else:
15:     Append rest of  $P[i : p]$  to  $R$ 
16:   return  $R$ .
```

Theorem 1. If P and Q are sorted, then $\text{COMBINE}(P, Q)$ returns a sorted array of the elements in P and Q .

Proof. We first show that R is sorted. The reason is that P and Q are sorted and thus elements added to R are increasing. Formally, in an iteration k of the while loop, an element is added to R as $R[k]$. This element is either an element $P[i]$ or an element $Q[j]$, for some i and j , whichever is smaller. Let us assume it was $P[i]$; the other case can be analogously argued. The previous element added to R (in the previous loop), that is, $R[k - 1]$ was either $P[i - 1]$ (in which case $i - 1$ was incremented to i) or it was $Q[j - 1]$. If the former, then $R[k - 1] \leq R[k]$ because $P[i - 1] \leq P[i]$ since P is sorted. If the latter, and this is crucial, then $Q[j - 1]$ must have been compared with $P[i]$ since i didn't increment in the $(k - 1)$ th loop. And thus, $Q[j - 1] \leq P[i] = R[k]$. That is, even in this case $R[k - 1] \leq R[k]$. Thus in each step, what is added to R is increasing implying R is sorted. Secondly, all elements of P and Q are visited in this order. And thus, R is a sorted order of all elements in P and Q . \square

Theorem 2. $\text{COMBINE}(P, Q)$ takes $O(p + q)$ time.

Proof. Here is a general principle: when analyzing the running-time of an algorithm with a *while loop*, one needs to figure out a measure/quantity/potential which (a) either monotonically strictly increases or strictly decreases in each iteration of the while loop, (b) starts off at a known value, and (c) one can argue termination of the while loop if the value ever reaches a different quantity. Almost *all* while loop running times are measured that way.

For the COMBINE , what is this quantity? Well, we see that in the while loop, either i increments or j increments. Therefore, the quantity of interest is $(i + j)$. This quantity starts off at 2. It always increases by 1. And finally note that if it ever reaches $(p + q + 2)$, then either $i \geq p + 1$ or $j \geq q + 1$ (for otherwise $i < p + 1, j < q + 1$ implying $i + j < p + q + 2$.) That is, the while loop terminates. This shows the while loop cannot have more than $(p + q)$ iterations. Since each iteration takes $O(1)$ time, we get the desired running time for COMBINE . \square

Armed with the above, the complete divide-and-conquer algorithm for sorting is given as:

```

1: procedure MERGESORT( $A[1 : n]$ ):
2:    $\triangleright$  Returns sorted order of  $A[1 : n]$ 
3:   if  $n = 1$  then:
4:     return  $A[1 : n]$ .  $\triangleright$  Singleton Array
5:    $m \leftarrow \lfloor n/2 \rfloor$ 
6:    $B_1 \leftarrow \text{MERGESORT}(A[1 : m])$ 
7:    $B_2 \leftarrow \text{MERGESORT}(A[m + 1 : n])$ 
8:   return  $\text{COMBINE}(B_1, B_2)$ 

```

Theorem 3. MERGESORT takes $O(n \log n)$ time.

Proof. Let $T(n)$ be the worst case running time of MERGESORT on arrays of size n . Since MERGESORT is a recursive algorithm, just as we did in Lecture 3 with MULT and DIVIDE , we try figuring out line by line the times taken. Let's begin.

First, we see that when $n = 1$, only [Line 3](#) and [Line 4](#) run. The time taken is $O(1)$. Thus, we get

$$T(1) = O(1) \tag{1}$$

Again, this is because *any* (finite) time algorithm on a constant sized input takes $O(1)$ time.

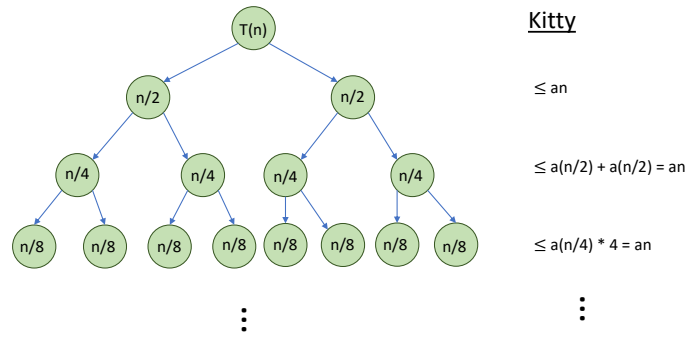
For larger n , the code proceeds to [Lines 6](#) to [8](#). [Line 6](#) is a recursive call. Since it is on an array of size m , by definition of $T()$, this takes *at most* $T(m)$ time. Similarly, [Line 7](#) takes at most $T(n - m)$ time. And, [Theorem 2](#) tells us that [Line 8](#) takes $O(n)$ time. Finally, we note that since $m = \lfloor n/2 \rfloor$, we get that $n - m = \lceil n/2 \rceil$. Therefore, putting all together, we get the recurrence.

$$T(n) \leq T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) + O(n), \quad \forall n > 1 \tag{2}$$

To get to the big picture, we get rid of the floors and ceilings. In the supplement, you can see why this is kosher². Furthermore, we also replace the $O(n)$ term by $\leq a \cdot n$ for some constant a , to get

$$T(n) \leq 2T(n/2) + a \cdot n, \quad \forall n > 1 \quad (3)$$

We will apply the “kitty method” or “opening up the brackets” method to solve the recurrence inequality given by (1) and (3).



$$\begin{aligned}
 T(n) &\leq 2T(n/2) + an \\
 &\leq 2(2T(n/4) + an/2) + an \\
 &= 4T(n/4) + 2an \\
 &\leq 4(2T(n/8) + an/4) + 2an \\
 &= 8T(n/8) + 3an \\
 &\vdots \\
 &\leq 2^k T(n/2^k) + kan
 \end{aligned}$$

Setting k such that $n/2^k \leq 1$ gives us $T(n) = O(n \log n)$. □

1.1 The Master Theorem

The following theorem is a useful hammer to solve many recurrences.

Theorem 4. Consider the following recurrence:

$$T(n) \leq a \cdot T(\lceil n/b \rceil) + O(n^d)$$

²if you are a little worried about this, then (a) good, and (b) note that for large x , $\lceil x \rceil$, $\lfloor x \rfloor$ are really $x \pm$ some “lower order” term, and so since we are talking using the Big-Oh notation, it shouldn’t matter. This is *not* a rigorous argument — it is not meant to completely convince you, but it should give you a hint to why this may be true.

where a, b, d are non-negative reals. Then, the solution to the above is given by

$$T(n) = \begin{cases} O(n^d) & \text{if } a < b^d \\ O(n^d \log n) & \text{if } a = b^d \\ O(n^{\log_b a}) & \text{if } a > b^d \end{cases}$$

The proof is similar in spirit to that of [Theorem 3](#) and you are recommended trying it. Instead of splitting into two, each “ball” splits into a different balls each of size n/b (ignoring floors & ceilings), but it puts $\Theta(n^d)$ in the kitty. If you write the expression as above (it’s a little more complicated than the one we say before), then you will get a *geometric series* whose base is exactly a/b^d . Thus if $a < b^d$ (that is, the base < 1), then the geometric sum is small, and the total cost is bounded by the first deposit in the kitty. If $a = b^d$ (that is, if the base $= 1$) then we make around the same amount of deposits in the kitty, and we do it $\Theta(\log n)$ times. Finally, if $a > b^d$, then the geometric series is bounded by the other end, and the “number of small balls” (which is this bizarrish term $n^{\log_b a}$) is what dominates. All this is perhaps too high-level to be useful – see the supplement for the proof.

```

1: procedure FUNNYSORT( $A[1 : n]$ ):
2:   if  $n = 1$  or  $n = 2$  then:
3:     return brute force sort  $A$  “in-place”
4:   else:
5:      $a \leftarrow \lceil n/3 \rceil$  and  $b \leftarrow \lceil 2n/3 \rceil$ 
6:     FUNNYSORT( $A[1 : b]$ )  $\triangleright$  “Sort” the first two third of the array
7:     FUNNYSORT( $A[a + 1 : n]$ )  $\triangleright$  “Sort” the last two third of the array
8:     FUNNYSORT( $A[1 : b]$ )  $\triangleright$  “Sort” the first two third of the array again
9:   return  $A$ 

```