CS31 (Algorithms), Spring 2020 : Lecture 3

Date:

Topic: Divide and Conquer, 1

Disclaimer: These notes have not gone through scrutiny and in all probability contain errors.
Please discuss in Piazza/email errors to deeparnab@dartmouth.edu

---

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.*

Before we go into the divide-and-conquer algorithm for sorting, let us discuss the naive algorithm. Here is one: scan the whole array to find the minimum element, set it to $B[1]$ and remove it from A. Repeat the above process $n - 1$ times more. The jth scan takes $(n - j)$ time, and thus the total time taken is $\Theta(n^2)$. Quadratic time is not bad; it is definitely better than going over all the $n!$ permutations and choosing the sorted one. But, we can 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 individually are sorted. That is why we can in fact 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

<u>COMBINE</u>
Input: Two sorted arrays $P[1:p]$ and $Q[1:q]$.
Output: Sorted array $R[1:r]$ 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. It takes $\Theta(p + q)$ time as each step takes $\Theta(1)$ time, in each step either i or j increments, and so the algorithm is over 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:     ▷ P and Q are sorted; outputs R a sorted array of elements in P and Q.
3:     i = j = k ← 1.
4:     while i < p + 1 and j < q + 1 do:
5:         if (P[i] ≤ Q[j]) then:
6:             R[k] ← P[i]
7:             i ← i + 1
8:         else:
9:             R[k] ← Q[j]
10:            j ← j + 1
           k ← k + 1
11:    if i > p then:
12:        Append rest Q[j : q] to R
13:    else:
14:        Append rest of P[i : p] to R
15:    return R.
```

Theorem 1. If P and Q are sorted, then 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. □

Theorem 2. 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 of 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. □

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

```
1: procedure MERGESORT(A[1 : n]):
2:     ▷ Returns sorted order of A[1 : n]
3:     if n = 1 then:
4:         return A[1 : n].  ▷ Singleton Array
5:     m ← ⌊n/2⌋
6:     B_1 ←MERGESORT(A[1 : m])
7:     B_2 ←MERGESORT(A[m + 1 : n])
8:     return 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 1 with MULT and DIVIDE, we try figuring out line by line the times taken. Except now, we are armed with the power of Big-Oh, and we can start putting stuff under that rug. Let's begin.

First, we see that when $n = 1$, only Lines 3 and 4 run. The time taken is some constant (independent of n) and we will call this unspecified constant $O(1)$. Unlike as we did in MULT and DIVIDE, note that we did not specify what time means (like BIT-ADDS or something). Whatever you choose, these lines take constant time. Thus, we get

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

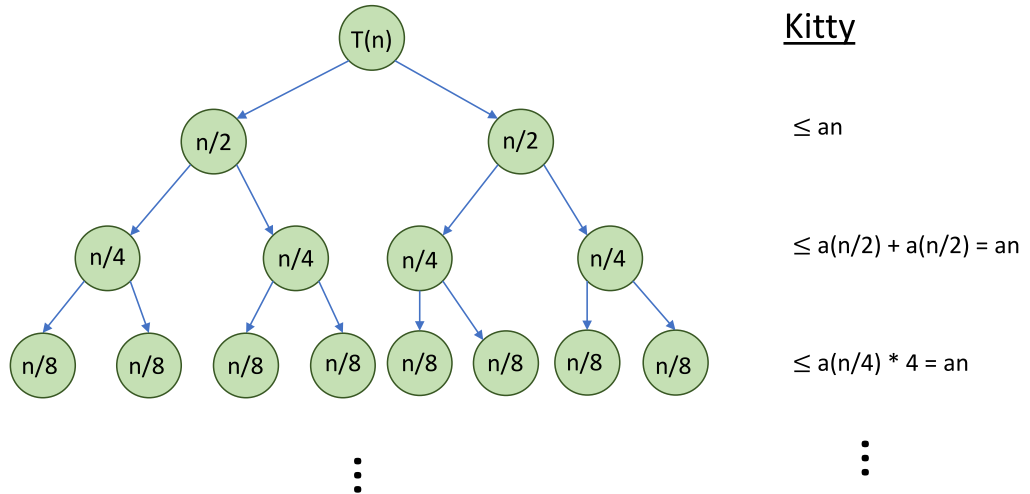
For larger n, the code proceeds to Lines 6,7, and 8. Line 6 is a recursive call. Since it is on an array of size m, by definition of $T()$, this takes *at most* $T(m)$ time. Similarly, Line 7 takes at most $T(n-m)$ time. And, Theorem 2 tells us that Line 8 takes $O(n)$ time. Finally, we note that since $m = \lfloor n/2 \rfloor$, 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 kosher[1]. Furthermore, we also replace the $O(n)$ term by $\leq a \cdot n$ for some constant a, to get

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

We will apply the "kitty method"



or "opening up the brackets" method to solve the recurrence inequality given by (1) and (3).

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

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

1.1 The Master Theorem

Theorem 4. Consider the following recurrence:

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

[1] if 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.

where a, b, d are non-negative integers. 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 quite similar to the proof that the recurrence (2) solves to $T(n) \le O(n \log n)$. 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. ✍

Exercise:

- *Solve the recurrence* $T(n) \le 2T(n/2) + \Theta(1)$.
- *Solve the recurrence* $T(n) \le T(n/3) + T(2n/3) + \Theta(n)$.
- *Solve the recurrence* $T(n) \le \sqrt{n} \cdot T(\sqrt{n}) + \Theta(n)$.

2 Counting Inversions

We now look at a closely related problem. Given an array $A[1:n]$, the pair (i, j) for $1 \le i < j \le n$ is called an *inversion* if $A[i] > A[j]$. For example, in the array $[10, 20, 30, 50, 40]$, the pair $(4, 5)$ is an inversion.

COUNTING INVERSION
Input: An array $A[1:n]$
Output: The number of inversions in A.
Size: n, the size of the array.

There is a naive $O(n^2)$ time algorithm: go over all pairs and check if they form an inversion or not. We now apply the divide-and-conquer paradigm to do better.

If $n = 1$, then the number of inversions is 0. Otherwise, suppose we divide the array into two: $A[1:n/2]$ and $A[n/2+1:n]$. Recursively, suppose we have computed the number of inversions in $A[1:n/2]$ and $A[n/2+1:n]$. Let these be I_1 and I_2, respectively. Note that any inversion (i, j) in $A[1:n]$ satisfies

(a) either $i < j \le n/2$, which implies (i, j) is an inversion in $A[1:n/2]$, or

(b) $n/2 + 1 \le i < j$, which implies (i, j) is an inversion in $A[n/2+1:n]$, or

(c) $i \le n/2 < j$, and these are the extra inversions over $I_1 + I_2$ that we need to count.

Let's call any (i, j) of type (c) above a *cross* inversion, and let C denote this number. Then by what we said above, we need to return $I_1 + I_2 + C$. Is it any easier to calculate C?

After you think about it for a while, there may not seem to be any easy way to calculate C faster than $O(n^2)$. There are two crucial observations that help here.

- The number of cross-inversions between $A[1:n/2]$ and $A[n/2+1:n]$ is the same as between $\mathsf{sort}(A[1:n/2])$ and $\mathsf{sort}(A[n/2+1:n])$, where $\mathsf{sort}(P)$ is the sorted order of an array P.
- If $A[1:n/2]$ and $A[n/2+1:n]$ were sorted, then the cross-inversions can be calculated in $O(n)$ time. This may not be immediate, but if you understand the COMBINE subroutine above, then it should ring a bell. We elaborate it on it later.

Cross-Inversions between Sorted Arrays. Given two sorted arrays $P[1:p]$ and $Q[1:q]$, we can count the number of cross-inversion pairs (i,j) such that $P[i] > Q[j]$ in $O(n)$ time as follows. As in COMBINE we start off with two pointers i,j initialized to 1. We also store a counter num initialized to 0. We check if $P[i] > Q[j]$ or not. If it isn't, that is if $P[i] \leq Q[j]$, then (i,j) is not a cross-inversion and we simply increment $i = i+1$. Otherwise, if $P[i] > Q[j]$, then we increment $\mathsf{num} = \mathsf{num} + (p-i+1)$ and $j = j+1$. Why so much? Well, not only is (i,j) a cross-inversion, so are $(i+1,j)$, $(i+2,j)$, and so on till (p,j). The claim below explains it more formally. We stop when either $i = p+1$ or $j = q+1$.

Claim 1. At any stage, suppose the algorithm encounters $P[i] > Q[j]$. Then $\{(i',j) : i' \geq i\}$ are the only cross-inversions which involve j.

Proof. Since P is sorted, $P[i'] > Q[j]$ for all $i' \geq i$ and so all such (i',j) are inversions. Now consider any $i'' < i$. Since in the algorithm the pointer is at $i > i''$, at some previous stage the algorithm compared $P[i'']$ and $Q[j'']$ with $j'' \leq j$, and found $P[i''] \leq Q[j'']$. But since Q is sorted, this would imply $P[i''] \leq Q[j]$. This implies for all $i'' < i$, (i'',j) is not an inversion. □

```
1: procedure COUNTCROSSINV(P[1:p], Q[1:q]):
2:     ▷ P and Q are sorted; outputs the number of (i,j) with P[i] > Q[j].
3:     i ← 1; j ← 1; num ← 0.
4:     while i < p+1 and j < q+1 do:
5:         if (P[i] > Q[j]) then:
6:             num ← num + (p − i + 1)
7:             j ← j + 1
8:         else:
9:             i ← i + 1
```

Theorem 5. COUNTCROSSINV counts the number of cross inversions between P and Q in time $O(p+q)$.

Now we are armed to describe the divide-and-conquer algorithm for counting inversions.

```
1: procedure CountInv1(A[1 : n]):
2:     ▷ Counts the number of inversions in A[1 : n]
3:     if n = 1 then:
4:         return 0. ▷ Singleton Array
5:     m ← ⌊n/2⌋
6:     I_1 ←CountInv1(A[1 : m])
7:     I_2 ←CountInv1(A[m + 1 : n])
8:     B_1 ←MergeSort(A[1 : m])
9:     B_2 ←MergeSort(A[m + 1 : n])
10:    C ←CountCrossInv(B_1, B_2)
11:    return I_1 + I_2 + C.
```

Let's analyze the time complexity. As always, let $T(n)$ be the worst case running time of CountInv1 on an array of length n. Let $A[1 : n]$ be the array attaining this time, and let's see the run of the algorithm on this array. The time taken by Lines 6 and 7 are $T(\lfloor n/2 \rfloor)$ and $T(\lceil n/2 \rceil)$ respecively. The time taken by Line 10 takes $O(n)$ time by what we described above. Furthermore, the Lines 8 and 9 takes $O(n \log n)$ time by Theorem 3. Together, we get the following recurrence

$$T(n) \le T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) + O(n \log n)$$

The above 'almost' looks like (2), and indeed the recurrence solves to $T(n) = O(n \log^2 n)$.

But there is something wasteful about the above algorithm. In particular, if you run it on a small example by hand you will see that you are sorting a lot. And often the same sub-arrays. Is there some way we can exploit this and get a faster algorithm?

Indeed we can. This introduces a new idea in the divide-and-conquer paradigm: *we can get more by asking for more*. This "asking for more" technique is something you may have seen while proving statements by induction where you can prove something you want by actually asking to prove something stronger by induction. In this problem, we ask our algorithm to do more: given an array $A[1 : n]$ it has to count the inversions and also has to sort the array too. Now note that in this case Lines 8 and 9 are not needed any more; this is returned by the new stronger algorithm. We however need to also return the sorted array : but this is what Combine precisely does. So the final algorithm for counting inversions is below.

```
1: procedure Sort-and-Count(A[1 : n]):
2:     ▷ Returns (B, I) where B = sort(A) and I is the number of inversions in A[1 : n]
3:     if n = 1 then:
4:         return (A, 0). ▷ Singleton Array
5:     m ← ⌊n/2⌋
6:     (B_1, I_1) ← Sort-and-Count(A[1 : m])
7:     (B_2, I_2) ← Sort-and-Count(A[m + 1 : n])
8:     C ← CountCrossInv(B_1, B_2)
9:     B ← Combine(B_1, B_2)
10:    return (B, I_1 + I_2 + C)
```

Now we see that the recurrence for the running time of Sort-and-Count is precisely

$$T(n) \le T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) + O(n)$$

Theorem 6. SORT-AND-COUNT returns the number of inversions of an array $A[1:n]$ in $O(n \log n)$ time.

As an application, you are now ready to solve Problem 1 of PSet 0. Go ahead and try it!