CS 3510 Algorithms 8/24/2023

Lecture 2: Merge Sort and Master Theorem

Lecturer: Abrahim Ladha *Scribe(s): Aditya Kumaran*

When do we use divide and conquer algorithms? These algorithms divide the larger problem into smaller, easier-to-solve subproblems, and use their solutions to help find a solution to the larger problem.

1 Merge Sort

Given an array of length n, we want to order the array such that they are sorted in increasing order. We assume here that all the elements are bounded, for easy comparisons.

Pseudocode :

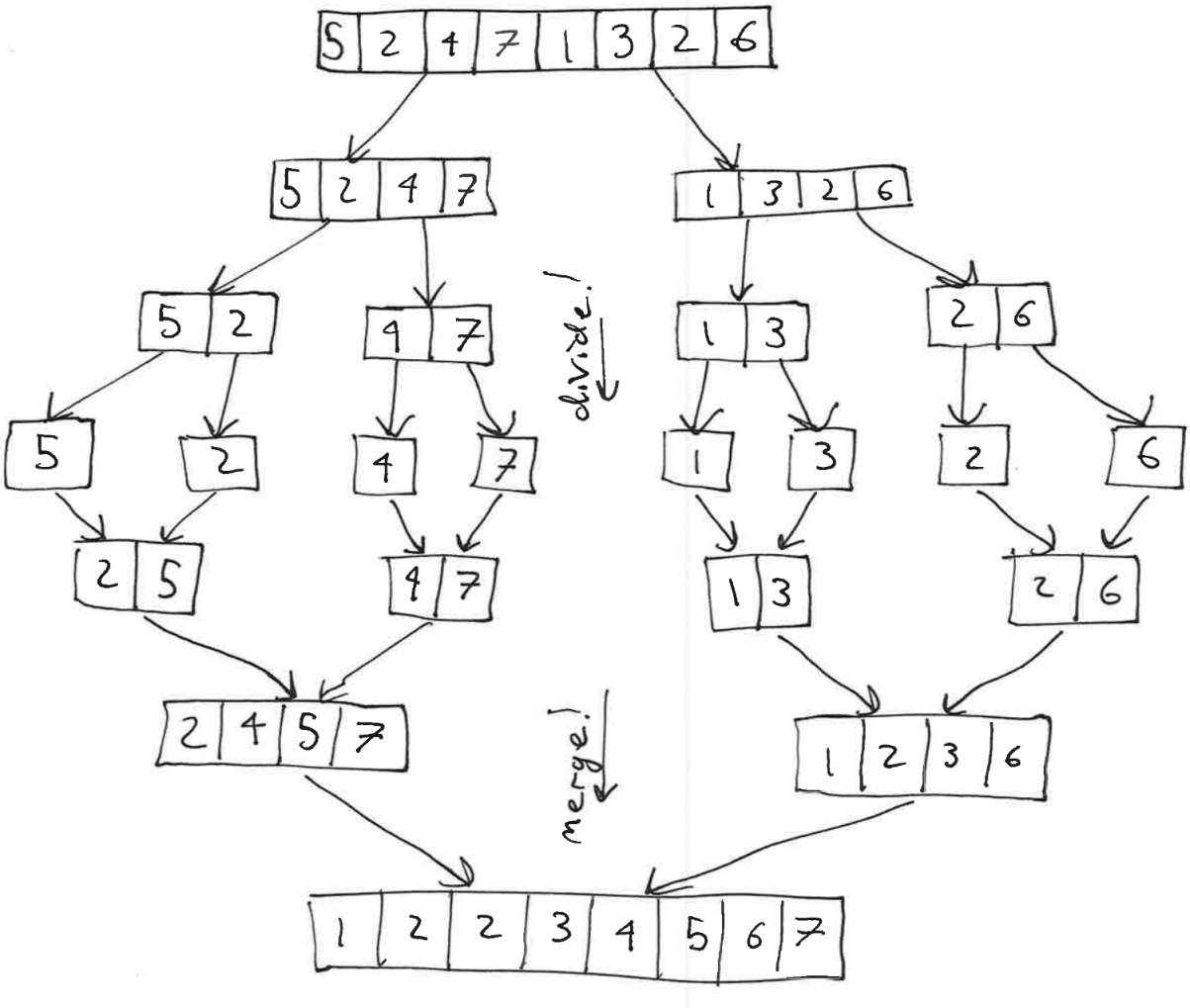
```
def mergesort(A[1..n]):
    ## Base Case
    if n == 1: return A
    ## Split problem into two subarrays, recursively call
   mergesort on both
    L = mergesort(A[1..floor(n/2)]
    R = mergesort(A[floor(n/2)+1..n]
    ## Return the merge of the left and right
    return merge(L,R)
```

Note : For odd length arrays, it doesn't matter which half the middle value is folded into, as long as you make a consistent definition for your algorithm.

Note : Arrays can be 0-indexed or 1-indexed depending on your own convention, as long as your pseudocode is consistent with your decided convention. Many textbooks use 1-indexing.

To prove the correctness of recursive algorithms, we typically use proof by induction. We can assume `merge()` is correct to prove correctness of `mergesort()`, and prove the correctness of `merge()` later.

Lets consider the execution on `A = [5,2,4,7,1,3,2,6]`



When `A` in `mergesort(A)` is length 1, mergesort returns the individual arrays to the parent call. Therefore, for each parent call (Ex.- when `A = [5,2]`) the values of L and R would be the individual values (Ex.- `L = [5], R = [2]`). As we move up the callstack, we call `merge()` on these two halves and return the sorted arrays (Ex.- `return = [2, 5]`). We can see here that the heavy lifting is being done by the recombining method, what we call `merge()`.

1.1 `merge()`

`merge()` takes two parameters - `x[1...k]` and `y[1...m]` We can assume that both these halves are also sorted, by induction - for the smallest case of length 2, our base case will handle the P(2) case in induction.

Pseudocode :

```
def merge(x[1...k], y[1...m]):
    if x is length 0
        return y
    if y is length 0
        return x
    if x[1] <= y[1]:
        return x[1].join(merge(x[2...k], y[1... k]))
    else:
        return y[1].join(merge(x[1...k], y[2... k]))
```

Note : It's very important that every divide and conquer algorithm has a base case, or it can recurse endlessly.

The beauty of this algorithm is that we are doing the smallest amount of work at every indivdual step, and combining this work in the most efficient way. Lets see if we can justify its correctness. Why does this merge the two sorted arrays? It is enough for us to argue that assuming `x,y` are sorted, that min(`x[1],y[1]`) is the first element of their sorted merger. Suppose without loss of generality that `x[1] < y[1]`. By assumption, note that `x[1]` is less than all of `x`, and `y[1]` is less than all of `y`. So we know since `x[1]` is less than the least element of `y`, that `x[1]` is less than all of `y`. So `x[1]` is less than all of the elements in both `x` and `y` meaning in the sorted merger, it should be the first element.

1.2 Example of how `merge()` works:

Let X = [2,4,5,7] and Y = [1,2,3,6], as in the penultimate stage of combination in the above mergesort callstack.

- We first compare `x[1]` and `y[1]`, here 2 and 1 respectively, and return `[1].join(merge([2,4,5,7],[2,3,6]))`
- We then compare 2 and 2, and return `[2].join(merge([4,5,7],[2,3,6]))`
- We then compare 4 and 2, and return `[2].join(merge([4,5,7],[3,6]))`
- We then compare 4 and 3, and return `[3].join(merge([4,5,7],[6]))`
- We then compare 4 and 6, and return `[4].join(merge([5,7],[6]))`
- We then compare 5 and 6, and return `[5].join(merge([7],[6]))`
- We then compare 7 and 6, and return `[6].join(merge([7],[]))`
- y is length 0, and return `[7].join([],[])`

The joins then return `[1,2,2,3,4,5,6,7]`, which is our solution.

`merge()` references every element in the array once, giving it a time complexity of $O(n)$.

But what is the time complexity of `mergesort()`?

- This is the time taken to solve a problem of size n => T(n).
- $T(n) = 2T(n/2)$ + time complexity of merge.
- Therefore, $T(n) = 2T(n/2) + O(n)$.

How do we analyze this recurrence to get the time complexity?
We can look at the recursion tree and count the leaves. Rather than solve this specific recurrence, we show a general way to solve recurrences of this form.

2 Master Theorem

Given a recurrence relation of the following form:

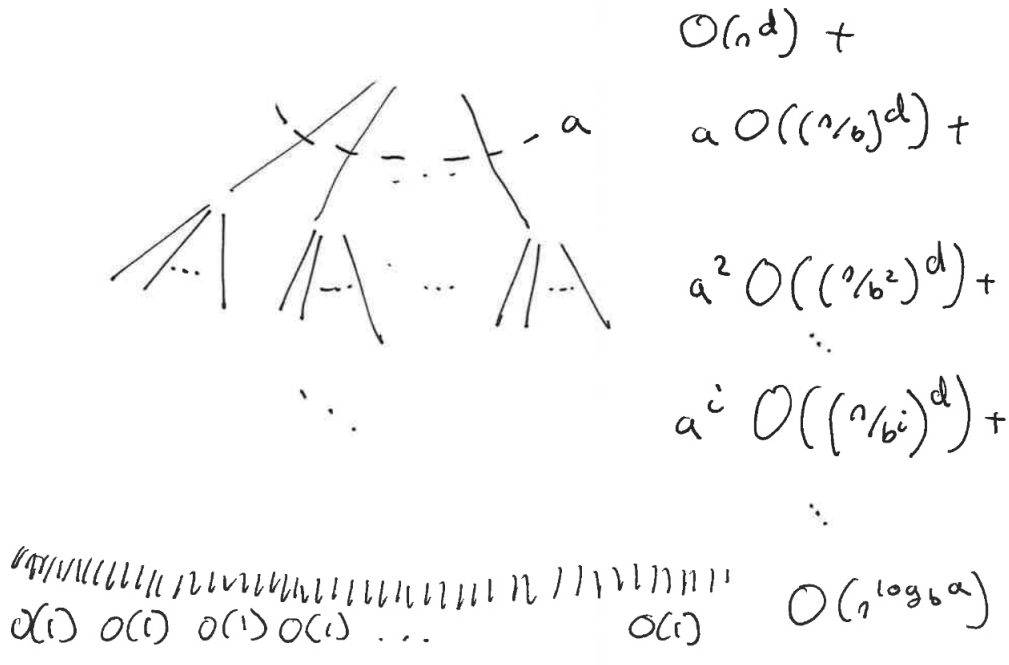
$$T(n) = aT(n/b) + O(n^d)$$

where

- a is the number of recursive calls
- b is the size of each subproblem (how many pieces are you dividing the problem into?)
- n^d is the time it takes to divide and recombine the problem.

Note : Usually, the time to divide the problem is negligible (python list slicing, etc.), and the recombination time makes up the majority of the n^d.

Consider the computation of a divide and conquer algorithm with such a general recurrence. We want to count the total work done. Think of a like the arity, or the number of branches, and think of b and d like the thickness of the next level of branches.



- At the top level, the subproblems work has already been completed, and only the final recombination needs to be done, so we see that the work done at this level is $O(n^d)$.
- At the next level, there are a sub problems, and the size of each subproblem has also been reduced to size n/b, so we see the work done at this level is $aO((n/b)^d)$
- At the next level, each of the previous a subproblems has a subproblem of their own, giving us a^2 subproblems. The size of the subproblem has been further divided to size n/b^2, giving the total work done at this level to be $a^2O((n/b^2)^d)$
- Continuing this like a geometric series, the work done at the following intermediate levels is $a^iO((n/b^i)^d)$
- The work done at the last level is the number of leaves times the work done at each leaf. Each leaf takes $O(1)$ to compute as a base case. The number of leaves we can compute from a little combinatorics. Given a binary tree of depth k, we see that it has 2^k leaves. Our tree has arity a so the number of leaves will be a^k where k is the depth. What is the depth of our tree? We continue to subdivide the problem until we divide out. That happens with (n/b^i) runs out. For what i does this happen? When $i = \log_b n$. Then the number of leaves is $a^{\log_b n} = n^{\log_b a}$. The work done at the level of the leaves is $O(1) \cdot (n^{\log_b a}) = O(n^{\log_b a)}$.
- We sum each level to get the total work done to be

$$T(n) = \sum_{i=0}^{\log_b n} a^i O\left(\left(\frac{n}{b^i}\right)^d\right) \; + \; O(n^{\log_b a})$$

- We want this in terms of big O, so we have three cases on what the dominating term is
- Case 1, if the work done at each exceeds how fast the problem subdivides, then the dominating term will be the first one of the summation in the series. This occurs when $d > \log_b a$ and the work done is $T(n) = O(n^d)$
- Case 2, if the work done at the leaves far exceeds the work required to recombine, the dominating term will be the number of leaves. Consider a tree with really thin branches but an insane amount of leaves. Each leaf may weigh nearly nothing but combined they are the heaviest part of this tree. This occurs when $d < \log_b a$ and the work done is $T(n) = \log_b a$.
- Case 3, if the work done subdivides quite neatly and doesn't increase or decrease one way or the other, we have to weigh the entire tree. The number of subproblems increases to a similar ratio as the size of the subproblems and the work done. This occurs when $d = \log_b a$. Every term of the sum is equivalent and we see then that $\sum_{i=0}^{\log_b n} a^iO((n/b^i)^d) + O(n^{\log_b a}) = \sum_{i=0}^{\log_b n} O(n^d) + O(n^{\log_b a}) = (\log_b n)O(n^d) + O(n^d) = O(n^d \log n)$

This gives us the final derivation of the master theorem. If $T(n) = aT(n/b) + O(n^d)$ then

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

Note : We don't write the base of logs when they aren't in the exponent because asymptotically (aka when concerned with big O and time complexity) they are equivalent. Ex- $O(\log_2(n)) = O(log\_3(n))$ In the exponent or when not "on the ground", the base of the log does matter.

3 Examples

We said that Mergesort has complexity from the recurrence: $T(n) = 2T(n/2) + O(n)$ With respect to the Master Theorem, here: $a = 2, b = 2, d = 1$. Using the Master Theorem cases above, we have $d = log_2(2) = 1$ Therefore, the time complexity of mergesort is $O(n^1 \log n) = O(nlogn)$. We could have computed this for the specific case of mergesort, but by doing it for a general recurrence, we can easily apply this to many problems.

As a sanity check, lets compute the run time of binary search, just to make sure that the master theorem works. You may have seen this implemented iteratively in the past, but it can be visualized recursively as well. We would split the whole array into 2 parts, and make a single recursive call on one of the halves, and the work done at each level is O(1) given that no real work is being done, we are just searching. Therefore, $T(n) = 1T(n/2) + O(1) = T(n/2) + O(1)$ We see that $a = 1, b = 2, d = 0$. Which case of the Master Theorem applies? Here d = $log_2(1) = 0$. Therefore, the time complexity of binary search is $O(n \log n) = O(logn)$. This is in fact the correct time complexity of binary search.
Note : We can see from this theorem that there are 3 ways to optimize a problem: reduce work at each level, reduce number of subproblems, reduce size of each subproblem. It is not uncommon for the number of subproblems to be reduced by making smarter ones, as we will see next time.