CS 3510 L2

Divide and Conquer is the first broad and battle technique of algorithm design. Today we will cover mergesort and the master theorem.

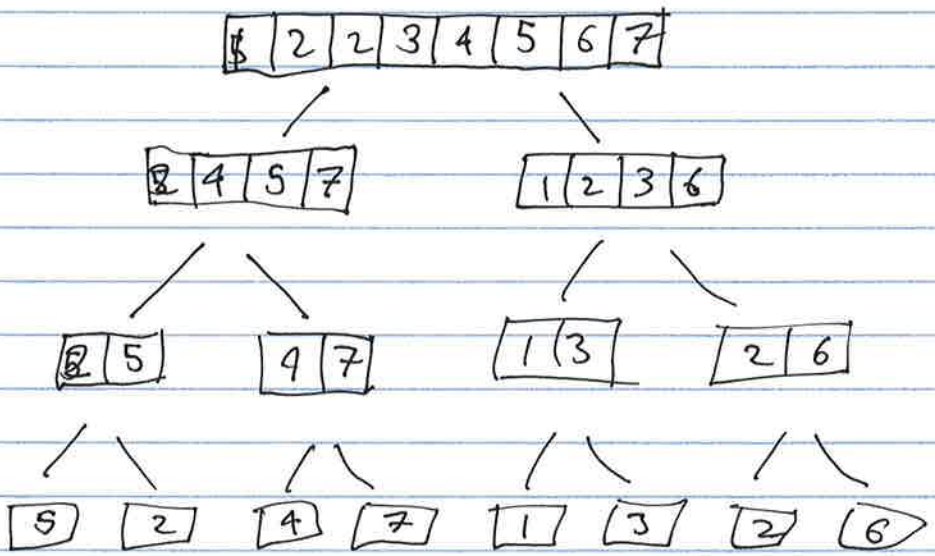
Divconquer is a technique where given a problem, you divide it into subproblems, make recursive calls to solve those, then recombine the subproblems. It is inherently recursive, and makes ample abuse of the call stack as a data structure.

Mergesort is a sorting algorithm. Given n comparable elements in an array of any order, it returns an array of them in sorted order. You perhaps have seen it before. It has two parts, first it splits the array of n elements into two sub arrays, then it merges two sorted arrays into one array. We discuss both problems individually.

```
def mergesort(A[1...n])
    if n = 1 return A
    L = mergesort(A[1... ⌊n/2⌋])
    R = mergesort(A[⌊n/2⌋+1 ... n])
    return merge(L, R)
```

First note how simple the algorithm looks. That is because all the work is being done really by this yet to be defined function called merge. This is common in divconquer, recursive calls make the code quite clean. Next note the obviousness of the correctness. By induction, as long as L, R are computed correctly, then mergesort is correct. For now lets assume the merge function does what we describe, and lets look at the execution of mergesort.

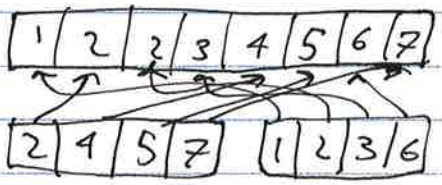
Suppose we want to sort [5, 2, 4, 7, 1, 3, 2, 6]



It also becomes clear how reliant it is on the merge algorithm. Lets solve that now. Given two sorted arrays, we want to merge them, also that the merger is also sorted. Think of like two piles, stacks, of sorted cards. Facing the two piles we take one are from the top, which ever is least one at a time. This ordering will be sorted since we choose the least. The guarantee that ours is sorted relies on the fact that the two inputs are also sorted. This leads to a natural recursive algorithm as well.

```
def merge(X[1..k], Y[1..l])
    if k=0 return Y
    if l=0 return X
    if X[1] ≤ Y[1]:
        return X[1].join(merge(X[2...k], Y[1...l]))
    else
        return Y[1].join(merge[X[1...k], Y[2..l])
```

Lets do an example. Consider merge([2,4,5,7],[1,2,3,6]).



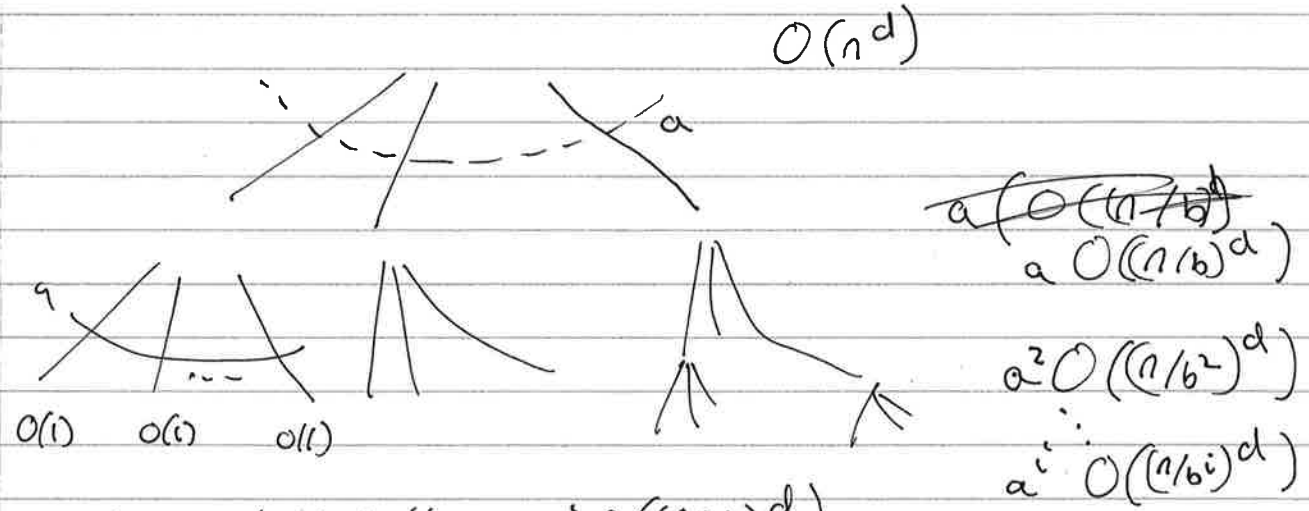
[2 4 5 7] [1 2 3 6]
[2 4 5 7] [2 3 6]
[4 5 7] [2 3 6]
[4 5 7] [3 6]
[4 5 7] [6]
[5 7] [6]
[7] [6]
[7] [ ]

Let us now analyze the run time of mergesort and merge. Let $T(n)$ be the time it takes to call mergesort on an array of n elements. It splits the problem into two subproblems, each of size $n/2$, then it calls merge to recombine them. Merge takes linear time, constant per each element so we see that $T(n) = 2T(n/2) + O(n)$.

Rather than solve this specific recurrence to get a number, we solve it in the general case. Consider a divide-and-conquer algorithm which takes $T(n) = aT(n/b) + O(n^d)$ where a is the number of subproblems, b is the size of each subproblem and $O(n^d)$ is the cost to both divide and recombine. In our mergesort recurrence, there are two subproblems of size $n/2$ and it takes $O(1)$ to divide and $O(n)$ time to recombine.

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

4



recombine cost at level i is $a^i O((n/b^i)^d)$

the tree has depth $\log_b n$, with all the leaves at the bottom. there are $a^{\log_b n} = n^{\log_b a}$ leaves. Each leaf takes $O(1)$ work so we see our total time is then

$$T(n) = O(n^{\log_b a}) + \sum_{i=0}^{\log_b(n)} a^i O((n/b^i)^d)$$

We want to present this nicer, so we split it up into three cases:

Case 1 $d > \log_b a$ the recombine cost outweighs the leaves. We take the largest term of the sum, the first, so $T(n) = O(n^d)$

Case 2 $d < \log_b a$ There ~~are~~ ~~are~~ ~~leaves~~ leaves than the recombine cost. Each is significant but the weight of the ones is much greater than the subproblems. The largest terms is then the leafs, so $T(n) = O(n^{\log_b a})$

Case 3 $d = \log_b a$ If the weight is balanced perfectly so each element of the summand is equivalent, we get

$$T(n) = O(n^{(\log_b a)}) + \sum^{\log_b n} O(n^d) = O(n^d) + \log n\, O(n^d) = O(n^d \log n)$$

Now lets apply this to mergesort. Recall our recurrence was $T(n) = 2T(n/2) + O(n)$. so $a=2$, $b=2$, $d=1$
Is d? $\log_b a$. $\log_2 2 = 1 = d$ so its case 2, well balanced. $T(n) = O(n^d \log n) = O(n \log n)$.

Heres another quick application, binary search. We all know it takes $O(\log n)$ but lets prove it by applying the master theorem. Binary search takes in n elements, makes a recursive call on half of them, and does no work at each level. Our recurrence is then $T(n) = 1 \cdot T(n/2) + O(1)$. where $a=1$, $b=2$, $d=0$ since $n^0 = 1$. Then $\log_2(1) = 0 = d$, so $T(n) = O(n^d \log n) = O(n^0 \log n) = O(\log n)$. Sanity check confirmed!

```
def binarysearch(A[1..n], t)
    if n=1 and A[1]=t
        return true
    else
        if A[⌊n/2⌋] > t
            return binarysearch(A[1...⌊n/2⌋], t)
        else
            return binarysearch(A[⌊n/2⌋+1, n], t)
```