Problem 1-1. (Constructing Recurrences)

(a) Construct the recurrence for the mergesort algorithm, and apply the master theorem to show that it is an $\Theta(n \log n)$ algorithm

(b) Construct the recurrence for the binary search algorithm, and apply the master theorem to show that it is an $\Theta(\log n)$ algorithm

Problem 1-2. the LIGHT side vs the DARK side (learning to interpret the master theorem). This question is meant for the students to gain a better intuition behind the Master Theorem's 3 cases

The following questions refer to the generalized recursion tree

(a) At each level j = 0,1,2,3.. $\log_b(n)$, there are how many subproblems?

(b) At each level j = 0,1,2,3.. $\log_b(n)$, what is the size of each subproblem?

(c) What is the total amount of work done at a given level j?

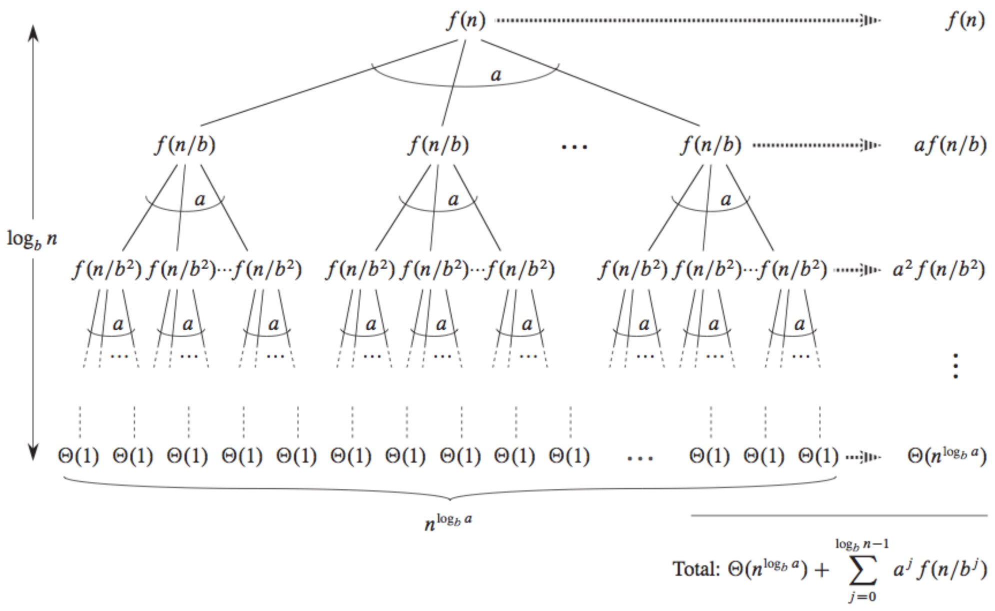
(d) When we compare the Rate of Subproblem Proliferations (dark side) to the Rate of Work Shrinkage (light side) there are three options:

1) RSP = RWS (the force is finally balanced!)

2) RSP < RWS

3) RSP > RWS

For each of these situations, explain where is most of the work going to be done (in the root level, at the leaves, or balanced work done at each level)



Problem 1-3. Find the overall asymptotic running time (i.e the value of $T(n)$) for the following recurrences either by the master theorem or by substitution:

(a) $T(n) = 7\,T(n/3) + n^2$

(b) $T(n) = 7\,T(n/2) + n^2$ (Strassen's method for matrix multiplication)

(c) $T(n) = 3\,T(n/2) + n^2$

(d) $T(n) = 2^n\,T(n/2) + n^n$

(e) $T(n) = 0.5T(n/2) + 1/n$

(f) $T(n) = 2T(n/2) + n$ (try with substitution as well)

Problem 1-4. (Quicksort)

(a) Define the recursion depth of QuickSort to be the maximum number of successive recursive calls before it hits the base case – equivalently, the number of levels in the recursion tree. Note that the recursion depth is a random variable, which depends on which pivots get chosen. What is the minimum-possible and maximum-possible recursion depth of QuickSort, respectively?

Problem 1-5. (Powers)

(a) Consider the following pseudocode for calculating a^b (where a and b are positive integers). What is the complexity of this algorithm?

```
function FastPower(a, b):
    if b = 1:
        return a
    else
        c = a * a
        ans = FastPower(c, b / 2)

    if b is odd:
        return a * ans
    else:
        return ans
end
```