Analyzing this Algorithm

- Recurrence relation will be used to analyze the runtime of this algorithm.
- $T(n)$ = the entire runtime of a problem of size n.
  - If n is small ($n \leq c$), we have a base case. (constant time) ($O(1)$)
  - Dividing each one of these larger problems into subproblems ($1/b$)

    $a = b = 2 \qquad T(n) = a\,T(n/b) + f(n)$
  - Time it takes to divide subproblems $= D(n)$
  - Combining the solutions $= C(n)$

$$T(n) = \begin{cases} O(1) & \text{if } n <= 1 \\ aT(n/b) + C(n) + D(n) & \text{otherwise} \end{cases}$$

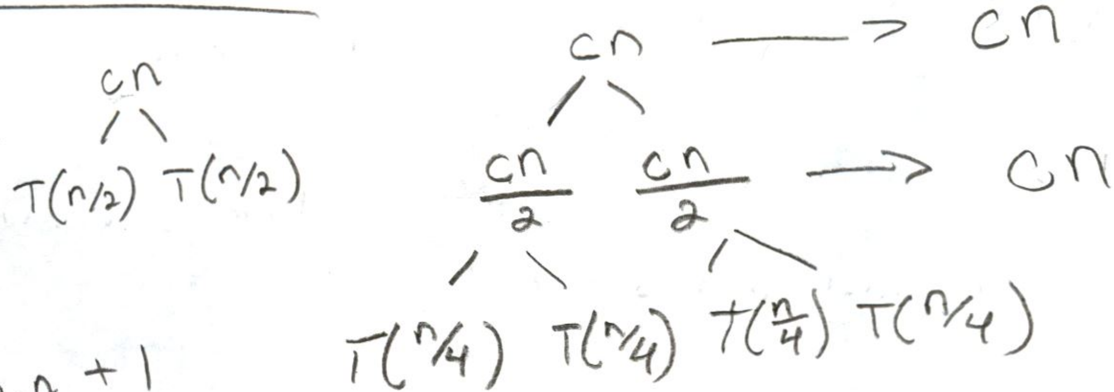
Mergesort

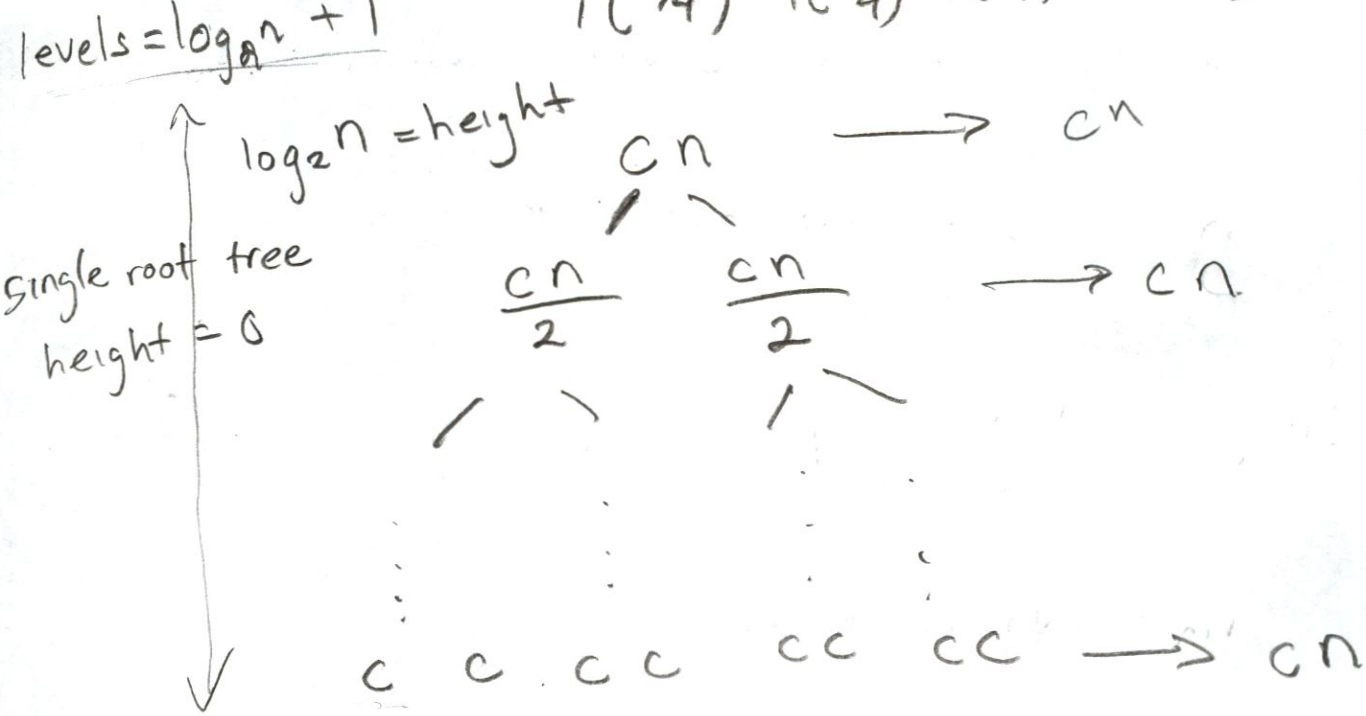
$$T(n) = \begin{cases} c & \text{if } n <= 1 \\ 2(T(n/2)) + \underline{cn} + c & \text{otherwise} \end{cases}$$

$$T(\tfrac{n}{2}) = \begin{cases} c & \text{if } n <= 1 \\ 2(T(n/4)) + c\tfrac{n}{2} + c \end{cases}$$

$$T(n) = \begin{cases} c & \text{if } n <= 1 \\ 2(2(T(\tfrac{n}{4}) + c\tfrac{n}{2} + c) + cn + c \end{cases}$$

Recursion Tree





Inductive proof

Base Case $n = 1$ $\qquad \log_2(1) + 1 = 1$ ✓

Inductive Step Assume problem size 2^i

$$\log_2 2^i + 1 = i + 1 + 1 = i + 2 \checkmark$$

$$\log_2 2^{i+1} + 1 = i + 1 + 1 = i + 2 \checkmark$$

Algorithm Runtime Complexity

Total Runtime = levels * amount of work in each level

Total Runtime = $(\log_2 n + 1)(cn)$

$$T(n) = O(n \log_2 n)$$

Space Complexity: $O(n)$

Quicksort

- Divide + conquer sorting algorithm
- Divide: Pick a "pivot" in the array. (an element, different ways to do this) Move the pivot into its final sorted location in the array by moving all elements less than the pivot to the left of the pivot + all elements greater than the pivot to the right of the pivot. Divide array into two subarrays, separated by the pivot.

Conquer: Recursively sorting the two arrays generated by the divide step.

Combine: No work is needed.

- Worst Case - $\Theta(n^2)$
- Expected Running Case - $\Theta(n \log n)$
  - Constants hidden in are small
- Space Complexity - $\Theta(1)$ in place.

Divide step step is implemented in a function called partition.

Pseudo Code

```
def Quicksort(A, start, end):
    if start < end:
        loc = partition(A, start, end)
        Quicksort(A, start, loc-1)
        Quicksort(A, loc+1, end)
```

Initial call? Quicksort(A, 0, len(A)-1)

$n = len(A) - 1 = end$

```
def partition(A, start, end):
    pivot = A[end]
    i = start - 1
    for j in range(start, end):
        if A[j] <= pivot:
            i = i+1
            swap(A[i], A[j])    <- defined elsewhere
    Swap(A[i+1], A[end])
    return i+1
```

Things to note ~~in pseudocode~~

- A[end] is always selected as our pivot.
- 4 different "regions" will need to be proved in our loop invariant.
  1. A[start .. i] $\leq$ pivot
  2. A[end] = pivot
  3. A[i+1 .. j-1] $\geq$ pivot
  4. A[j .. end-1] ← unexamined region.