Merge Sort

The merge sort algorithm deals with the problem of sorting a list of n elements. It is able to sort a list of n elements in $O(n \log n)$ runtime, which is considerably faster than insertion sort, which takes $O(n^2)$. Merge sort uses a divide and conquer method:

1. If the length of the list is 1, the list is sorted. Return the list
2. Otherwise, split the list in two (roughly) equal halves and then recursively merge sort the two halves
3. Merge the two sorted halves into one sorted list

The merge operation takes two sorted lists and an iterator at the head of each list. At each step, we compare the elements at the iterators with each other. We take the smaller element, add it to our merged list, and then advance the iterator associated with that smaller element. We repeat this step until every element in the two sorted halves has been added to the merged list, forming a single large sorted list. The runtime of the merge operation is $O(n)$ where n is the number of elements we are merging.

Solving Recurrences

We can use merge sort as an example of how to solve recurrences. Recall back to peak finding where we solved recurrences by showing them in the form of "Runtime of original problem" = "Runtime of reduced problem" + "Time taken to reduce problem", and then solved them using the dot dot dot method. We are going to formalize this a little more.

The form of the recurrences that we'll be dealing with look like:

$$T(n) = aT(n/b) + f(n) \tag{1}$$

Where a is the number of subproblems that the original problem is divided into, n/b is the size of each subproblem, and $f(n)$ is how long it takes to divide into subproblems and combine the results of the subproblems.

Example: Binary Search Binary searching a sorted list involves splitting the list in two and recursively searching into one half of the list. $a = 1$ since whenever we split the list in two, we just call a binary search on one half. $b = 2$ since the new subproblem has half the elements of the original problem. $f(n) = O(1)$ since finding the middle of a list and deciding which half to recurse into is a constant time operation.

Example: Merge Sort Merge sorting a list involves splitting the list in two and recursively merge sorting each half of the list. $a = 2$ since we call merge sort twice at every recursion (once on each half). $b = 2$ since each of the new subproblem has half the elements in the original list. $f(n) = O(n)$ since combining the results of the subproblems, the merge operation, is $O(n)$.

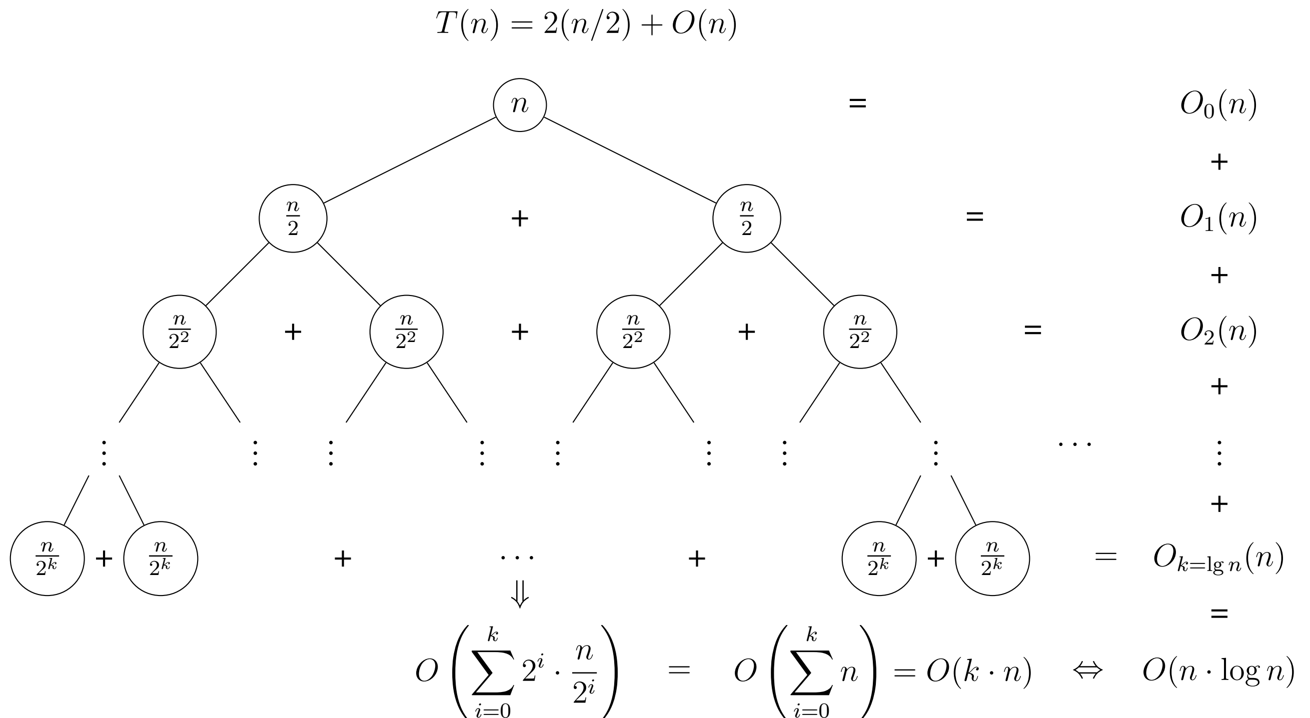
Tree Method

One way to solve recurrences is to draw a recursion tree where each node in the tree represents a subproblem and the value at each node represents the amount of work spent at each subproblem. The root node represents the original problem. In a recursion tree, every node that is not a leaf has a children, representing the number of subproblems it is splitting into. To figure out how much work is being spent at each subproblem, first find the size of the subproblem with the help of b, then substitute the size of the subproblem in the recurrence formula $T(n)$, then take the value of $f(n)$ as the amount of work spent at that subproblem. Long story short, a node with a problem size of x, the node will have a children each contributing $f(x/b)$ amount of work.

The work at the leaves is $T(1)$, since at that point we have divided the original problem up until it can no longer be further divided. Note that this means that the work contributed by the leaves are $O(1)$.

Once we have our tree, the total runtime can be calculated by summing up the work contributed by all of the nodes. We can do this by summing up the work at each level of the tree, then summing up the levels of the tree.

Example: Merge Sort Recursion Tree



The merge sort recursion tree is considered somewhat balanced. The work done at each level stays consistent (in this case, it is $O(n)$ at each level) so the total work done can be calculated by multiplying the work at each level by the number of levels, hence $O(n \log n)$ running time for merge sort. However, there are two other cases that may happen. If the work at each level

geometrically decreases as we go down the tree, then the work done at the root node (i.e. $f(n)$) will dominate the runtime. If instead the work at each level geometrically increases as we go down the tree, then the work done at the bottom level (i.e. number of leaves $\times f(1)$ or O(number of leaves) or $O(n^{\log_b a})$) will dominate the runtime.

Master Theorem

Whether or not the work per level stays relatively consistent, geometrically increases, or geometrically decreases can be determined by looking at a, b, and $f(n)$ from the recurrence formula. Using this information gives us the master theorem, which allows us to solve recurrences of the form $T(n) = a(n/b) + f(n)$.

1. **Case 1:** If $f(n) = \Theta(n^{\log_b a - \epsilon})$, then the amount of work per level geometrically increases as we go down the tree. The work at the leaf level dominates and $T(n) = \Theta(n^{\log_b a})$.

2. **Case 2:** If $f(n) = \Theta(n^{\log_b a} \log^k n)$, then the amount of work per level have about the same cost (i.e. work per level does not *polynomially* increase or decrease, though work per level still may increase or decrease at some slower rate). We have to take the work of all levels into account and $T(n) = \Theta(n^{\log_b a} \log^k n \log n) = \Theta(n^{\log_b a} \log^{k+1} n)$.

3. **Case 3:** If $f(n) = \Theta(n^{\log_b a + \epsilon})$, then the amount of work per level geometrically decreases as we go down the tree. The work at the root level dominates and $T(n) = \Theta(f(n))$.

Example 1: $T(n) = 2T(n/2) + 1$

Example 2: $T(n) = 2T(n/2) + n$

Example 3: $T(n) = 2T(n/2) + n \log n$

Example 4: $T(n) = 4T(n/2) + n^3$