Recursive Algorithms Analysis

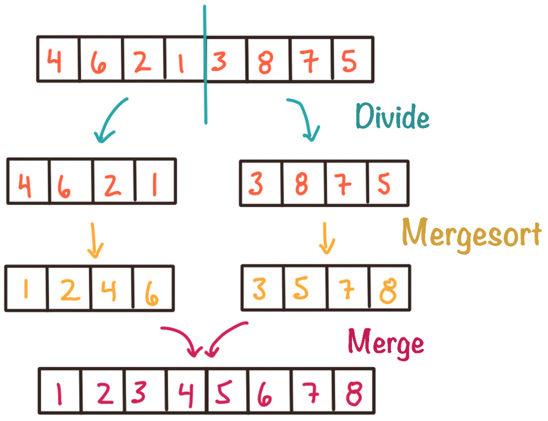
In this lecture, we analyze algorithms that follow a **design-and-conquer** approach. Our first example is Mergesort, a very well known recursive algorithm. We show three main techniques to analyze the run-time of these problems:

- Use a recurrence tree
- Guess a solution and then prove it works by induction
- Use the Master Method

1 Mergesort

In our first lecture, we defined the sorting problem. In this section we look at the **Mergesort** algorithm, which sorts an array of n items using a divide-and-conquer approach. The three steps of mergesort can be described as follows:

- **Divide** the array of length n into two subsections of size $n/2$.
- **Conquer:** Sort the two smaller sections using a *recursive* call to mergesort on *each half*.
- **Combine:** *Merge* the two smaller sections together into one large sorted sequence.

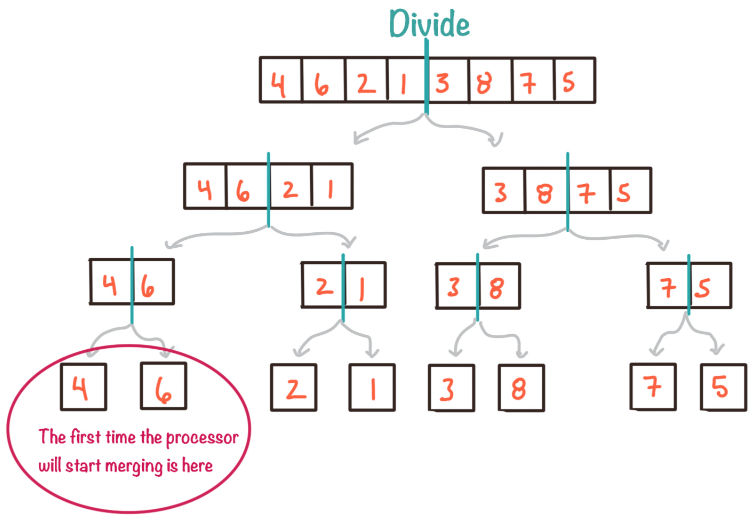


It is the *conquer* step above that is the **magic or recursion**: our algorithm for merge-sort actually *calls itself*, but on an array of half the size. It may seem a bit daunting that we assume a computer knows how to do something that we are currently teaching it how to do. When a recursive call is made to the same algorithm, the execution then turns to completing that recursive call before returning to the rest of the original procedure. An execution stack is keeping track of a sort of "to-do list" of what it needs to finish before returning to where it was.

1.1 The execution of Merge-sort

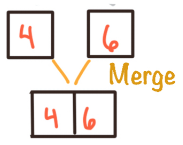
Assume that we receive as input to merge-sort an array A indexed between s and f. Certainly if A has only one element in it, then the procedure can return immediately: a single element is sorted by default. Otherwise, we recursively sort the left and right half of the array, and then merge the results together. The recursive calls are made to arrays that are getting smaller and smaller. Eventually, a call to merge-sort will be made on an array that is very small: one with only one element in it. These calls terminate quickly, since an array of size one is already sorted. Only when the recursive calls to *each half* of the array terminate can the execution then proceed with the merge step.

We trace through the execution of merge-sort below, on an array of size 8. This execution is also animated in the class videos. In the first step, the array is divided in half. Our algorithm calls merge-sort on *each* half, each of those calls will also need to divide the array in half, etc. So no merge process is carried out until each of those halves has completely finished. This means the array is continuously divided in half, until at some point, the length of the array is *one*. At this point, the merge-sort algorithm returns the single element as "*sorted*". In the figure below, we can see how the array is divided into smaller and smaller sections until each piece contains only 1 number.

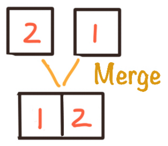


The call to merge-sort on an array of size 1 terminates. In the above example, the red circle section would be merged into the sorted list.

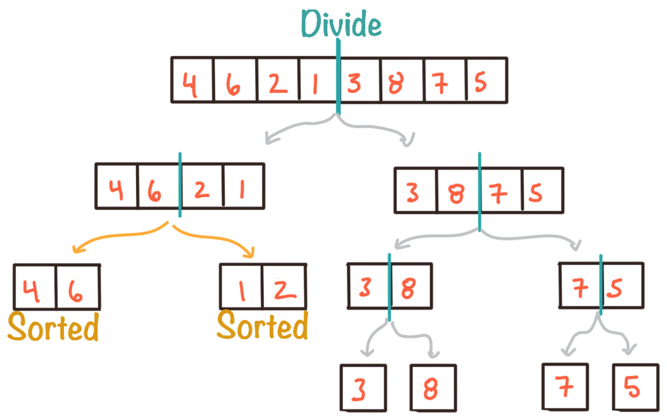
The first section to get merged:



Next, the arrays containing 2 and 1 would be merged:



At this stage, the diagram looks like this:



Notice that the rest of the tasks in the tree are just "waiting" their turn. The next job would be to merge (4,6) and (1,2) together, completing the entire left side of the tree before moving onto the right side. In order to fully understand the *order* of the operations, I suggest the following online tool: https://www.hackerearth.com/practice/algorithms/sorting/merge-sort/visualize/

1.2 The Merge-Sort pseudocode

The digram above shows the "behind-the-scenes" work that is actually carried out during the execution of merge-sort. However, the actual *instructions* to merge-sort are quite simple. All the work encompassed in the continuous division of the array is carried out by the two recursive calls to merge-sort. The pseudocode is shown below. Again, we assume we receive as input an array A indexed from s to f. The algorithm terminates with the elements sorted in the original array.

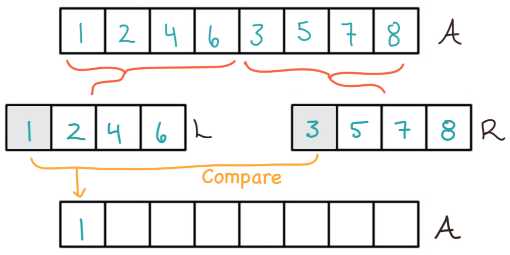
```
MergeSort($A$, $s$,$f$)

    if $s < f$
        $q = \lfloor \frac{s+f}{2} \rfloor$
        MergeSort($A, s, q$)
        MergeSort($A, q+1, f$)
        Merge($A, s, q, f$)
```

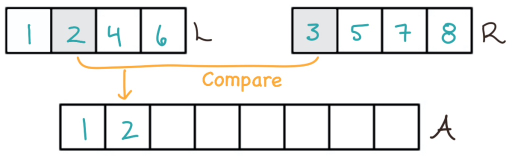
1.3 Combine: The Merge step of Merge-sort

The *Merge* procedure is a step of the Merge-sort algorithm. We first illustrate the procedure and then look at the detailed pseudocode.

The Merge step receives as input one array $A[s, \ldots, f]$ where the first and second half are each sorted. The first step is to copy each of these halves into new arrays, called $L[]$ and $R[]$. Next, we compare the first elements of each array - the smaller of the two is selected and "*placed*" into the front position of the new merged list.



The "front" of the list L is now incremented. Next, we repeat this comparison with the next two front elements from each list.



This process continues until all elements have been moved over into the new merged list. Our class video animates this process.

The pseudocode for Merge-sort is shown below. As input, the procedure takes in the array A which is sorted between indices s and q and between indices $q+1$ and f. The result is that the two subarrays are merged and copied back into the final array A between indices s and f. As usual, our arrays below are assumed to be indexed at 1.

```
Merge($A$, $s$, $q$, $f$)

    Let $L[]$ and $R[]$ be new arrays
    Copy elements from $A$ between $s$ and $q$ into array $L[]$
    Set $L[q-s+2] = \infty$
    Copy elements from $A$ between $q+1$ and $f$ into array $R[]$
    Set $R[f-q+1] = \infty$
    $i = 1, j = 1$
    for $k = s$ to $f$ do:
        if $L[i] < R[j]$
            $A[k] = L[i]$
                $i++$
            else $A[k] = R[j]$
                $j++$
```

How long does this take? The copying of the arrays takes $\Theta(n)$ time. Notice that we carry out exactly **one comparison** each time we move **one element** over to the merged array, A. Since we must move over exactly n elements, the merge procedure takes $\Theta(n)$ operations.

1.4 The running time

Let $T(n)$ be a function that represents the running time of mergesort when the array has size n. When the array has size 1, mergesort will simply return the array as-is, because a single element is always considered sorted. This takes constant time, and suppose that constant is c_1. Thus

$$T(1) = c_1$$

For recursive algorithms we determine the running time $T(n)$ for $n > 1$ by focusing on how many operations are carried out at each phase of the divide and conquer process:

- Divide: it takes constant time to "cut" an array in half
- Conquer: Each subarray has size $n/2$, and so takes time $T(n/2)$ to sort. In total, that is $2T(n/2)$.
- Merge: To merge the two halves together takes $\Theta(n)$. In order to express this without the Θ notation, assume the merge operation takes cn steps for some constant c.

From the above list, we have that

$$T(n) = 2T(n/2) + cn$$
$$T(1) = c_1$$

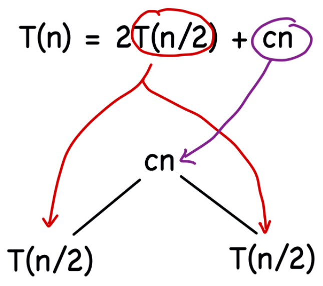
This is called a recursive equation for the running time $T(n)$, since it depends on $T(n/2)$. The goal is to find a solution to this equation - something for which we can simply plug-in an n value and get out a value for $T(n)$.

2 The recursion tree

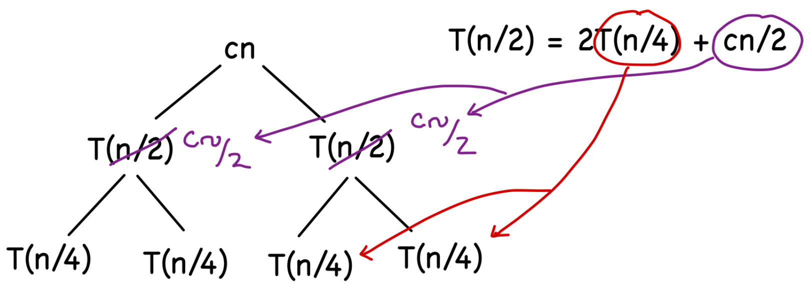
In this section we look at our first method of solving the above recurrence, using a recursion tree. This is a structure that visually represents how many operations are carried out during the different steps of the recursive algorithm. We demonstrate the recursion tree for the mergesort recurrence:

$$T(n) = 2T(\frac{n}{2}) + cn$$

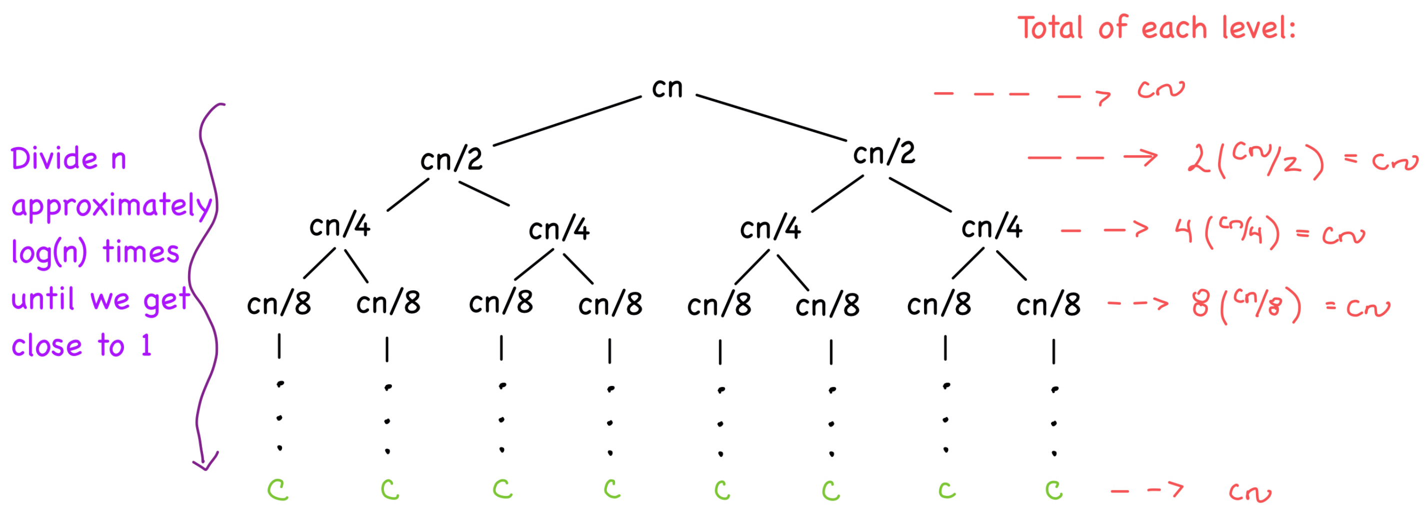
The root of the tree is where we place the cost of the *merge* operation, and below the root we place the cost of carrying out mergesort on each half:



After this, we *replace* $T(n/2)$ in the next level with the same procedure: in its spot we put the cost of merging and below that, the cost of mergesort of each half of the $n/2$ array.



This process continues. Eventually, we will have divided n by 2 enough times that we end up with an input size of just one element. Suppose for simplicity that n is an exact power of 2. Then if we divide n by 2 exactly $\log_2 n$ times then we end up with 1. As we already mentioned, $T(1) = c_1$, so the final tree looks like:



We are now ready to use the tree to determine the run time of mergesort:

- The **total run-time** of mergesort is the **total** of all the values in this tree.
- Let's look at the **total cost per level**. Notice that the sum of all the values in each level is exactly cn. Since there are $\log_2 n$ levels with a total value of cn, then the total time needed for all these levels is:

$$cn \log_2 n$$

$$c_1 n$$

- The total runtime of mergesort is:

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

This expression represents the run-time of mergesort, expressed in a way that is not recursive. This function is $\Theta(n \log n)$. Thus we have the final run time of mergesort in asymptotic notation.

The recursion tree method works well if there is a pattern that we can identify easily when we sum the number of operations in each level.

3 Substitution

The above method relies heavily on the fact that we are able to notice a pattern in the number of operations per level of the tree, and that it is easy to sum these values together. In this section, we look at a second method for solving a recurrence. Suppose we have a **guess** of what the solution is. We could **prove** that this solution is indeed **correct** using **induction**. Even if your induction is rusty, focus on the steps of the following example that prove the solution to $T(n) = 2T(n/2) + cn$ is $O(n \log n)$.

Example 1. *Use substitition to prove that the mergesort recurrence has solution $O(n \log n)$*

Solution:

Step 1: The first step is to have an idea of what the runtime is. Since we showed above that the runtime of mergesort was $O(n \log n)$ using the recursion tree, then we use that "guess" here. In order to *prove* that this fact is true, we need to satisfy the definition of $O(n \log n)$, and so the job now is to achieve:

Goal: Show that $T(n) \leq dn \log n$

for some constant d.

Step 2:. Substitution

- **Assumption:** Induction works by *assuming* the statement is true for small values of n and then proving this implies it works for larger values. The small values work like this: imagine that you replace n with $\frac{n}{2}$ in the statement we are trying to prove. Then *assume* that

$$T(\frac{n}{2}) \leq d\frac{n}{2}\log(\frac{n}{2})$$

- **Substition:** Now we *substitute* this into our recurrence equation an hope that this *proves* that $T(n) \leq dn \log n$.

$$\begin{aligned}
T(n) &= 2 \cdot T(\frac{n}{2}) + cn \\
&\leq 2 \cdot d\frac{n}{2}\log(\frac{n}{2}) + cn \\
&= dn \log n - dn \log 2 + cn \\
&= dn \log n - n(d - c) \\
&\leq dn \log n \text{ as long as } d \geq c
\end{aligned}$$

This second-last line subtracts off $n(d-c)$, making the whole thing less than $dn \log n$ as long as $d - c \geq 0$ or in other words if $d \geq c$. So we have just shown that $T(n) \leq dn \log n$ for all large enough n and therefore $T(n)$ is $O(n \log n)$.

4 Master Method

In this final section, we look at a method that solves recurrences of a specific form. In these recurrences, the algorithm takes a problem of size n and *calls* itself on a problem of size $\frac{n}{b}$. So as long as $b > 1$, it is working on a smaller problem size. Master method can be applied to any recurrence of the following form:

$$T(n) = aT(\frac{n}{b}) + f(n)$$

For example, the recurrence $T(n) = 2T(\frac{n}{3}) + n^3$ falls into this category, where $a = 2$, $b = 3$ and $f(n) = n^3$. The master method provides us with the run time of these algorithms *directly*, without having to work with the recursion tree or induction. We simply need to determine the relationship between a, b and $f(n)$, and then decide which of the following three cases we are in:

Master Method
Suppose $T(n) = aT(\frac{n}{b}) + f(n)$. Determine the value of: $k = \log_b a$. The master method **compares the function** $f(n)$ **to the function** n^k: (assume that the constant e is a small positive constant).

- **Case 1:** Suppose that the function $f(n)$ is asymptotically smaller than n^k. This happens when $f(n)$ is $O(n^{k-e})$. In this case, $T(n)$ is $\Theta(n^k)$.
- **Case 2:**. Suppose that the function $f(n)$ is asymptotically the same as n^k. This happens when $f(n)$ is $\Theta(n^k)$. In this case, $T(n)$ is $\Theta(n^k \log n)$.
- **Case 3:** Suppose the function $f(n)$ is asymptotically larger than n^k. This happens when $f(n)$ is $\Omega(n^{k+e})$. In this case, $T(n)$ is $\Theta(f(n))$.

Example 2. *Apply the master theorem to*

$$T(n) = 8T(n/2) + n^2$$

Solution: In this example, $a = 8$, $b = 2$ and $f(n) = n^2$. Then $k = \log_b a = \log_2 8 = 3$. Now we just need to compare n^3 and $f(n) = n^2$. Certainly, the dominant of the two is n^3. Since $f(n) = n^2$ is $O(n^{3-e})$ we are in case 1 of the master method and we can conclude that $T(n)$ is $\Theta(n^3)$.

Example 3. *Apply the master theorem to*

$$T(n) = 7T(n/2) + n^2$$

Solution: Since $a = 7$ and $b = 2$, we have that $k = \log_2 7 = 2.807$. Thus we need to compare $n^{2.807}$ to the function $f(n) = n^2$. In this case, $n^{2.807}$ seems dominant, and in fact, $f(n)$ is $O(n^{2.807-e})$ and so by case 1, $T(n)$ is $\Theta(n^{2.807})$.

Example 4. *Apply the master theorem to:*

$$T(n) = 4T(\frac{n}{4}) + 2n$$

Solution: Since $a = 4$ and $b = 4$, then $k = \log_4 4 = 1$. Then we compare $f(n) = 2n$ to n^1. Since $f(n)$ is $\Theta(n)$, then we conclude by case 2 above that $T(n)$ is $\Theta(n \log n)$.