Recurrence Relations & The Master Theorem

Suhas Arehalli

COMP221 - Spring 2026

1 Analyzing Runtimes of Recursive Functions

Our techniques thus far taught us to analyze runtime by first computing growth functions (roughly) and then applying the appropriate simplifications to express that function in Big-O notation. While this technique is universal, we quickly get into trouble when analyzing certain kinds of recursive functions. Consider MergeSort in Alg. 1.

Algorithm 1 Mergesort, with Merge's structure implied. See the MergeSort notes for details on Merge!

```
function MergeSort(Array A[1...n])
    if N ≤ 1 then
        return A
    end if
    m ← ⌈n/2⌉
    L ← MergeSort(A[1...m])
    R ← MergeSort(A[m+1...n])
    return Merge(L, R)
end function
```

If we apply our standard tools, we see that the cost of a call to MergeSort is a bunch of $\Theta(1)$ work, a $\Theta(n)$ call to Merge, and... two recursive calls to MergeSort, each on an input roughly half the size of the full array.

It might initially be confusing what to do with these recursive calls! If we don't know the runtime complexity of MergeSort, we can't know what the recursive calls cost in time complexity and thus... can't determine the runtime complexity of MergeSort. Our first new tool — recurrence relations — are meant just to get us proper notation to talk about this!

Let $T(n)$ denote the worst-case growth function for MergeSort. We can express the runtime of MergeSort as

$$T(n) = \begin{cases} 2T(\frac{n}{2}) + \Theta(n), & n > 1 \\ \Theta(1), & n \leq 1 \end{cases}$$

Note a few things about my notation here. First, it's a *recurrence relation*: we allow for $T(n)$ to appear on both sides (i.e., $T(n)$ is defined in terms of itself). This matches the recursive nature of the algorithm! Additionally, note that (unlike Erickson) I set up the recurrence as a piecewise function, making explicit that there is both a recursive and base case. This is what I consider good analytic hygiene — this is tedious, but both technically correct and good at reminding you of the structure of the problem (we're trying to approach a base case!). Third, worth reminding you of our shorthand for Big-O: It is common to write, for instance, $\Theta(g(n))$ in the place of some function $f(n) \in \Theta(g(n))$. Finally, our cue to do more work, is to note that this is not a nice *closed-form* — it doesn't look like something we can, for instance, turn into Big-O notation! We need to write a non-recursive definition of $T(n)$!

1.1 Unrolling Recurrences

The first approach is to *unroll* the recursive part of the recurrence until you deduce a pattern. To unroll, we just substitute the equation into itself over and over, noting any patterns with respect to the number of substitutions.

$$
\begin{aligned}
T(n) &= 2T(\frac{n}{2}) + \Theta(n) \\
&= 2(2T(\frac{n}{4}) + \Theta(\frac{n}{2})) + \Theta(n) \\
&= 4T(\frac{n}{4}) + 2\Theta(\frac{n}{2}) + \Theta(n) \\
&= 4(2T(\frac{n}{8}) + \Theta(\frac{n}{4})) + 2\Theta(\frac{n}{2}) + \Theta(n) \\
&= 8T(\frac{n}{8}) + 4\Theta(\frac{n}{4}) + 2\Theta(\frac{n}{2}) + \Theta(n) \\
&= 2^3 T\left(\frac{n}{2^3}\right) + 2^2\Theta(\frac{n}{2^2}) + 2^1\Theta(\frac{n}{2^1}) + 2^0\Theta(\frac{n}{2^0})
\end{aligned}
$$

So after k substitutions we find

$$
T(n) = 2^{k+1}T\left(\frac{n}{2^{k+1}}\right) + \sum_{i=0}^{k} 2^i\Theta\left(\frac{n}{2^i}\right)
$$

And have a hope of getting rid of the T term on the right hand side — we simply need to choose k such that $T(\frac{n}{2^{k+1}})$ is a base case! This requires $\frac{n}{2^{k+1}} \leq 1$, Which solves out to $k \geq \log n - 1$, and so we want $k = \log n - 1$ and we can substitute $\Theta(1)$ for our call to T, getting

$$
\begin{aligned}
T(n) &= 2^{\log n}\Theta(1) + \sum_{i=0}^{\log n - 1} 2^i\Theta\left(\frac{n}{2^i}\right) \\
&= n\Theta(1) + \sum_{i=0}^{\log n - 1} 2^i\Theta\left(\frac{n}{2^i}\right) \\
&= \Theta(n) + \sum_{i=0}^{\log n - 1} \Theta(n) \\
&= \log n \cdot \Theta(n) = \Theta(n \log n)
\end{aligned}
$$

And we can conclude that MergeSort is $\Theta(n \log n)$ as desired!

We can do the same for something like Binary Search! Consider Alg. 2

Algorithm 2 Binary Search

```
function BinarySearch(Array A[1...n], target e)
    if n == 0 then
        return NULL
    end if
    m ← ⌈n/2⌉
    if A[m] == e then
        return m
    else if e ≤ A[m] then
        return BinarySearch(A[1...m − 1], e)
    else
        return BinarySearch(A[m + 1...n], e)
    end if
end function
```

Whose recurrence relation for worst-case runtime will be

$$T(n) = \begin{cases} T(\frac{n}{2}) + \Theta(1), & N > 0 \\ \Theta(1), & N = 0 \end{cases}$$

And then we can unroll, getting

$$\begin{aligned} T(n) &= T(\frac{n}{2}) + \Theta(1) \\ &= (T(\frac{n}{4}) + \Theta(1)) + \Theta(1) \\ &= ((T(\frac{n}{8}) + \Theta(1)) + \Theta(1)) + \Theta(1) \end{aligned}$$

And so for k substitutions, we get

$$T(n) = T(\frac{n}{2^k}) + k\Theta(1)$$

And to get to our base case, we need $k = \log n$, getting

$$\begin{aligned} T(n) &= T(1) + (\log n)\Theta(1) \\ &= \Theta(1) + \log n \cdot \Theta(1) \\ &= \Theta(\log n) \end{aligned}$$

As expected. Always reassuring when new methods give us the answers we know are correct a priori!

1.1.1 More formally...

It's worth noting that we're being a sketchy with our use of Big-Θ notation here — Why are are we saving the $\frac{1}{2^i}$ terms in the summation when we know that they are constant factors? Why do we wait to cancel out the 2^i term later on? This can be clarified by getting rid of the shorthand and just working with a proper growth function. Feel free to skim this section more lightly than the others.

If we let $f(n) \in \Theta(n)$ be the non-recursive cost of MergeSort, then our unrolling analysis goes as follows:

$$\begin{aligned}
T(n) &= 2T(\frac{n}{2}) + f(n) \\
&= 2(2T(\frac{n}{4}) + f(\frac{n}{2})) + f(n) \\
&= 4T(\frac{n}{4}) + 2f(\frac{n}{2}) + f(n) \\
&= 4(2T(\frac{n}{8}) + f(\frac{n}{4})) + 2f(\frac{n}{2}) + f(n) \\
&= 8T(\frac{n}{8}) + 4T(\frac{n}{4}) + 2f(\frac{n}{2}) + f(n)
\end{aligned}$$

And we see the pattern that after k substitutions, we get

$$T(n) = 2^{k+1}T(\frac{n}{2^{k+1}}) + \sum_{i=0}^{k} 2^i f(\frac{n}{2^i})$$

And substitute $k = \log n - 1$, getting

$$\begin{aligned}
T(n) &= 2^{\log n}T(\frac{n}{2^{\log n}}) + \sum_{i=0}^{\log n - 1} 2^i f(\frac{n}{2^i}) \\
&= nT(1) + \sum_{i=0}^{\log n - 1} 2^i f(\frac{n}{2^i})
\end{aligned}$$

Hopefully this looks similar to the more informal analysis above, but with an explicit function in the place of the informal Θ notation.

Now, we know that since $f(n) \in \Theta(n)$, There exist $c > 0$ and n_0 such that $c_1 n \leq f(n) \leq c_2 n$ for $n \geq n_0$. Then we have that:

$$\sum_{i=0}^{\log n-1} 2^i(c_1\frac{n}{2^i}) \le \sum_{i=0}^{\log n-1} 2^i f(\frac{n}{2^i}) \le \sum_{i=0}^{\log n-1} 2^i(c_2\frac{n}{2^i})$$
$$\sum_{i=0}^{\log n-1} c_1 n \le \sum_{i=0}^{\log n-1} 2^i f(\frac{n}{2^i}) \le \sum_{i=0}^{\log n-1} c_2 n$$
$$(\log n)c_1 n \le \sum_{i=0}^{\log n-1} 2^i f(\frac{n}{2^i}) \le (\log n)c_2 n$$
$$c_1 n\log n \le \sum_{i=0}^{\log n-1} 2^i f(\frac{n}{2^i}) \le c_2 n\log n$$

And thus we know that $\sum_{i=0}^{\log n-1} 2^i f(\frac{n}{2^i}) \in \Theta(n\log n)$ by definition, with the same $c_1, c_2 > 0$ and the same n_0! Then, it is hopefully less fraught that $nT(1) \in \Theta(n)$ (as $T(1) \in \Theta(1)$), and so:

$$T(n) = nT(1) + \sum_{i=0}^{\log n-1} 2^i f(\frac{n}{2^i}) \in \Theta(n\log n)$$

1.1.2 Where informality goes wrong

Here, I take some time to diagnose why it's dangerous to simplify $\Theta(\frac{n}{2^i})$ in the shorthand to $\Theta(n)$. Feel free to skim this even more lightly!

What we first need to remember is that $\Theta(n)$ is actually shorthand for a function that runs in linear time — some $f(n)$ such that $f(n) \in \Theta(n)$. When I write $\Theta(\frac{n}{2})$, there is a slight shift in meaning — by substituting $\frac{n}{2}$ for n, I'm not saying I have a function that runs in $\Theta(\frac{n}{2})$ time, but that I'm running an $\Theta(n)$ function on an input of size $\frac{n}{2}$. But just because they're slightly different doesn't mean they're not equivalent! If we rewrite that same statement formally, we're saying that $f(n) \in \Theta(n)$, then $g(n) = f(\frac{n}{2}) \in \Theta(n)$. This is not too bad: If $f(n) \in \Theta(n)$, then we know there is a $c_1, c_2 > 0$ and n_0 such that $n \ge n$

$$\begin{aligned} c_1 n &\le f(n) \le c_2 n \\ c_1\frac{n}{2} &\le f(\frac{n}{2}) \le c_2\frac{n}{2} \\ \frac{c_1}{2}n &\le f(\frac{n}{2}) \le \frac{c_2}{2}n \end{aligned}$$

And so with $\frac{c_1}{2}, \frac{c_2}{2}$ and n_0 we have $g(n) = f(\frac{n}{2}) \in \Theta(n)$. This is, of course, able to be extended to *any* $f(\frac{n}{k})$ for any fixed k.

Surely this means that $\Theta(\frac{n}{k}) = \Theta(n)$ substitutions are reasonable in our simplified notation as before! And yet, if we do those in our old analysis:

$$
\begin{aligned}
T(n) &= n\Theta(1) + \sum_{i=0}^{\log n - 1} 2^i \Theta(\frac{n}{2^i}) \\
&= \Theta(n) + \sum_{i=0}^{\log n - 1} 2^i \Theta(n) \\
&= \Theta(n) + \Theta(n) \sum_{i=0}^{\log n - 1} 2^i \\
&= \Theta(n) + \Theta(n) \cdot 2^{\log n} = \Theta(n^2)
\end{aligned}
$$

Which is wrong! Our more careful analysis showed that it's wrong! but where is the mistake? It is subtle, but the trick is that the index i obscures the dependence between the denominator 2^i and the n (the upper bound on our sum!).

Here is a simpler scenario to ponder to help you realize the issue: First, we start with $f(\frac{n}{k}) \in \Theta(n)$ for any $k \in \mathbb{N}$ if $f(n) \in \Theta(n)$ — this is known to be correct! However, we cannot let $k = n$ in this identity — our intuition should tell us this needs to be $\Theta(1)$! Why not? because n is not fixed — it's the argument to f! In the same way, we cannot get rid of the 2^i because the values i takes depends on n (as it's the upper bound for the summation!). For another piece of evidence, consider the last term in the sum: in that term, 2^i becomes $2^{\log n - 1} = \frac{n}{2}$ — clearly depends on n! Same for the term before, where it will take the value $\frac{n}{4}$! These are *not* constant factors that can be ignored! At the very least, this should be an indication that we should go back to the formal definitions where the shorthand is lacking.

1.2 Recurrence Trees

Another method to analyze the runtime of recursive functions is to study a *recurrence tree*, a tree structure where each node represents a call to the function and its children denote the calls to it's children. We annotate each node with the cost of executing that call *excluding work done by the recursive calls*. In that way, the cost of a call including the recursive call is the sum of the annotations of the corresponding node and all its descendants. Thus, the cost of calling the first, non-recursive call is the sum of the annotations to all nodes in the tree!

Take, for example, MergeSort. We observe that in the first call, we do $\Theta(n)$ work outside of calls, and make two recursive calls, so we draw a tree with a root labeled $\Theta(n)$ to indicate the work done by the first call, and give it two children.

Those children will do $\Theta(\frac{n}{2})$ work (dominated by a call of size $\frac{n}{2}$ to Merge), and each make two calls, and so on until a base case is reached. We can then identify a couple of features of the tree we can measure: the total amount of work done at each level of the tree (In this case, 2^k calls each doing $\Theta(\frac{n}{2^k})$ work), and the height of the tree (Since the tree splits the problem size in half each level, roughly $\log_2 n$ levels to get to our base case). This gives us Fig. 1.

From these features, we can compute the sum of the annotations on all nodes, summing level-by

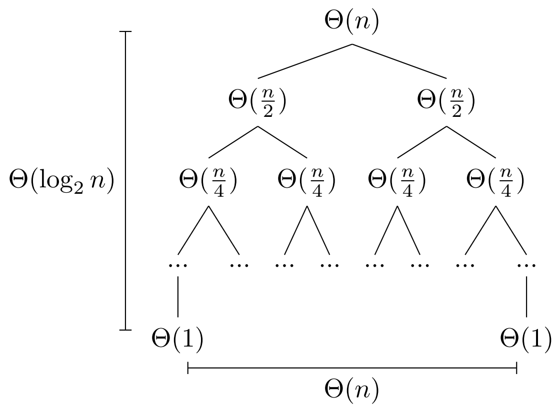

Figure 1: A recursion tree for a call to MergeSort of size n

level

$$\begin{aligned} T(n) &= \sum_{k=0}^{\log n} 2^k \Theta(\frac{n}{2^k}) \\ &= \log n \cdot \Theta(n) = \Theta(n \log n) \end{aligned}$$

Converging with our prior result.

As you can see, both of these informal techniques converge on not only the same result, but the same eventual summation. The perk of seeing both is that they take different angles on what this the same mathematics: unrolling is a bit more directly math-y, and models the recursion via substitution. The recurrence trees help visualize the call structure and give you a way to see how the terms we gather in unrolling correspond to recursive depth in the call structure. The recurrence trees also help us give language to dynamics that emerge in different classes of algorithms. These patterns are captured in what is called the *Master Theorem.*

2 The Master Theorem

Theorem 2.1 (The Master Theorem)**.** *Given a recurrence relation*

$$T(n) = \begin{cases} aT(\frac{n}{b}) + f(n), & n \geq k \\ \Theta(1), & n < k \end{cases}$$

where $a, b \in \mathbb{N}$*,*

1. *If* $f(n) \in \Theta(n^{\log_b a})$*, then* $T(n) \in \Theta(n^{\log_b a} \log n)$
2. *If* $f(n) \in O(n^c)$*, with* $c < \log_b a$*, then* $T(n) \in \Theta(n^{\log_b a})$
3. *If* $f(n) \in \Omega(n^c)$ *with* $c > \log_b a$ *and* $af(\frac{n}{b}) \leq kf(n)$ *for* $0 < k < 1$*, then* $T(n) = \Theta(f(n))$

To understand the Master Theorem, it's important to get a handle on the value $n^{\log_b a}$, which shows up in each case! This is the number of leaves (i.e., base cases!) in the recursive tree formed by a call of size n. Observe that this is actually just

$$\begin{aligned} n^{\log_b a} &= a^{\log_a (n^{\log_b a})} \\ &= a^{\log_b a \log_a n} \\ &= a^{\log_b n} \end{aligned}$$

Then note that a is the *branching factor* of the tree (how many children each node will have) and $\log_b n$ is the depth of our recursive tree!

Then we should note that since each base case takes $\Theta(1)$ time to complete, this means that just solving all of the base cases takes $\Theta(n^{\log_b a})$ time! The key dynamic here is whether our summation is dominated by the cost of solving the base cases, doing non-recursive work in each call, or whether these are balanced against each other.

There recurrence tree can help us give each of these cases a visual:

1. In Case 1, we have something that looks like MergeSort! The amount of work done at each node ($f(n)$) is proportional to the amount of work done by all of the leaves in it's the subtree ($\Theta(n^{\log_b a})$, and so the amount of work done at each level will be proportional! In this case, our total runtime is the work done at each level ($\Theta(n^{\log_b a})$) multiplied by the number of levels ($\Theta(\log n)$)!

2. In Case 2, the work done at each node ($f(n)$) is *strictly*[1] less than the work done by the leaves in it's subtree. This means that the work done in the tree is dominated by work done at the leaves! Thus, our runtime complexity is simply the cost of handling all of our base cases: $\Theta(n^{\log_b a})$!

3. Case 3 is a bit more complex. Here, we have that the work done at each node strictly dominates the work done by the leaves it generates. This alone gets us nowhere though, since this just tells us that the leaves won't dominate and nothing about how to sort out all of the internal layers! We need an additional assumption: That for any given node, the work done by it's children ($af(\frac{n}{b})$) is only some fraction k of the work done at that node itself ($f(n)$).

   Together, this information tells us that the root's time complexity dominates: There aren't that many base cases, and the work at lower levels is just some fraction of work at a higher level, so our time complexity is just the work done at the highest level, $\Theta(f(n))$!

3 Analyzing the Master Theorem in Generality

Consider the generic recursion seen in the Master Theorem:

$$T(n) = \begin{cases} aT(\frac{n}{b}) + f(n), & n > k \\ \Theta(1), & n \leq k \end{cases}$$

[1]Note that this is the purpose of c! The exponent of n must be strictly smaller than $\log_b a$!

Through a recursion tree or unrolling analysis, we can see that at recursion depth 0, we have $f(n)$ work done. At depth 1, we have a recursive calls each doing $f(\frac{n}{b})$ work. At depth 2, we have a^2 calls (a from each of the a calls on the prior level) doing $f(\frac{n}{b^2})$ work, and so on. The height, and thus maximum depth, will be $\log_b n$, getting us

$$\begin{aligned} T(n) &= \sum_{i=0}^{\log_b n-1} a^i f(\frac{n}{b^i}) + a^{\log_b n}\Theta(1) \\ &= \sum_{i=0}^{\log_b n-1} a^i f(\frac{n}{b^i}) + \Theta(a^{\log_b n}) \\ &= \sum_{i=0}^{\log_b n-1} a^i f(\frac{n}{b^i}) + \Theta(n^{\log_b a}) \end{aligned}$$

Our analysis hinges on the relationship between these two terms — the base case/leaf term on the right and the summation representing work in all of the recursive calls. Each of the cases gives us additional assumptions, so let's see how this plays out in each case. **While none of this is beyond your ability, you shouldn't feel responsible for this material! It is purely here for reference!**

3.1 Case 1: $f(n) \in \Theta(n^{\log_b a})$

In this case, we apply the definition of Big-Θ to analyze the summation $S(n) = \sum_{i=0}^{\log_b n-1} a^i f(\frac{n}{b^i})$ with the given $c_1, c_2 > 0$ and n_0 s.t. for all $n \geq n_0$:

$$\begin{gathered} \sum_{i=0}^{\log_b n-1} a^i c_1 (\frac{n}{b^i})^{\log_b a} \leq S(n) \leq \sum_{i=0}^{\log_b n-1} a^i c_2 (\frac{n}{b^i})^{\log_b a} \\ \sum_{i=0}^{\log_b n-1} a^i c_1 (\frac{n^{\log_b a}}{b^{i \log_b a}}) \leq S(n) \leq \sum_{i=0}^{\log_b n-1} a^i c_2 (\frac{n^{\log_b a}}{b^{i \log_b a}}) \\ \sum_{i=0}^{\log_b n-1} a^i c_1 (\frac{n^{\log_b a}}{a^i}) \leq S(n) \leq \sum_{i=0}^{\log_b n-1} a^i c_2 (\frac{n^{\log_b a}}{a^i}) \\ \sum_{i=0}^{\log_b n-1} c_1 n^{\log_b a} \leq S(n) \leq \sum_{i=0}^{\log_b n-1} c_2 n^{\log_b a} \\ c_1 n^{\log_b a} \log_b n \leq S(n) \leq c_2 n^{\log_b a} \log_b n \end{gathered}$$

Which tells us that the summation $S(n) \in \Theta(n^{\log_b a} \log n)$, and so

$$\begin{aligned} T(n) &= S(n) + \Theta(n^{\log_b a}) \\ &= \Theta(n^{\log_b a} \log n) + \Theta(n^{\log_b a}) \in \Theta(n^{\log_b a} \log n) \end{aligned}$$

As desired.

3.2 Case 2: $f(n) \in O(n^c)$ for $c < \log_b a$

First, let $\varepsilon = \log_b a - c > 0$, and thus $c = \log_b a - \varepsilon$. Then, applying our definition of Big-O to $S(n)$ as applied above, we find that for $k > 0$ and n_0 with $n \geq n_0$

$$\begin{aligned}
S(n) &\leq \sum_{i=0}^{\log_b n-1} a^i k(\frac{n}{b^i})^c \\
&\leq \sum_{i=0}^{\log_b n-1} k(\frac{a^i}{b^{ic}})n^c \\
&\leq \sum_{i=0}^{\log_b n-1} k(\frac{a^i}{b^{i(\log_b a-\varepsilon)}})n^c \\
&\leq \sum_{i=0}^{\log_b n-1} k(\frac{a^i}{a^i b^{-i\varepsilon}})n^c \\
&\leq \sum_{i=0}^{\log_b n-1} k(b^{i\varepsilon})n^c \\
&\leq \left(\sum_{i=0}^{\log_b n-1} (b^\varepsilon)^i\right) kn^c
\end{aligned}$$

Now just apply the formula for solving a geometric series to find

$$\begin{aligned}
S(n) &\leq \left(\frac{1-(b^\varepsilon)^{\log_b n}}{1-b^\varepsilon}\right) kn^c \\
&\leq \left(\frac{1-n^\varepsilon}{1-b^\varepsilon}\right) kn^c \\
&\leq \frac{k}{1-b^\varepsilon}(n^c - n^{\log_b a})
\end{aligned}$$

Now, we should observe that the denominator, $1 - b^\varepsilon$ is negative: As $b > 1$, $b^\varepsilon > 1$! So, lets pull out and distribute that negative term, then drop the remaining negative term

$$S(n) \leq \frac{k}{b^\varepsilon - 1}(n^{\log_b a} - n^c) \leq \frac{k}{b^\varepsilon - 1}(n^{\log_b a})$$

Now we see with $\frac{k}{b^\varepsilon-1}$ and n_0 we have that $S(n) \in O(n^{\log_b a})$, and thus

$$\begin{aligned}
T(n) &= S(n) + \Theta(n^{\log_b a}) \\
&= O(n^{\log_b a}) + \Theta(n^{\log_b a}) \in \Theta(n^{\log_b a})
\end{aligned}$$

3.3 Case 3: $f(n) \in \Omega(n^c)$ with $c > \log_b a$ and $af(\frac{n}{b}) \leq kf(n)$ for $0 < k < 1$

Again, let's analyze $S(n)$:

$$S(n) = \sum_{i=0}^{\log_b n-1} a^i f(\frac{n}{b^i})$$

We then apply the $af(\frac{n}{b}) \leq kf(n)$ inequality inductively, getting

$$\begin{aligned} S(n) = \sum_{i=0}^{\log_b n-1} a^i f(\frac{n}{b^i}) &\leq \sum_{i=0}^{\log_b n-1} k^i f(n) \\ &\leq \left(\frac{1-k^{\log_b n}}{1-k}\right) f(n) \leq \frac{1}{1-k} f(n) \end{aligned}$$

And thus with $\frac{1}{1-k}$ and n_0, we have that $S(n) \in O(f(n))$.
We can also derive the Ω and thus conclude $S(n) \in \Theta(f(n))$ by noting that

$$\begin{aligned} S(n) &= \sum_{i=0}^{\log_b n-1} a^i f(\frac{n}{b^i}) \\ &= f(n) + \sum_{i=1}^{\log_b n-1} a^i f(\frac{n}{b^i}) \geq f(n) \end{aligned}$$

Now we can use the fact that $f(n) \in \Omega(n^c)$ for $c > \log_b a$ to see that $f(n)$ dominates $n^{\log_b a}$, and then we can say

$$\begin{aligned} T(n) &= S(n) + \Theta(n^{\log_b a}) \\ &= \Theta(f(n)) + \Theta(n^{\log_b a}) \in \Theta(f(n)) \end{aligned}$$

And we're done!