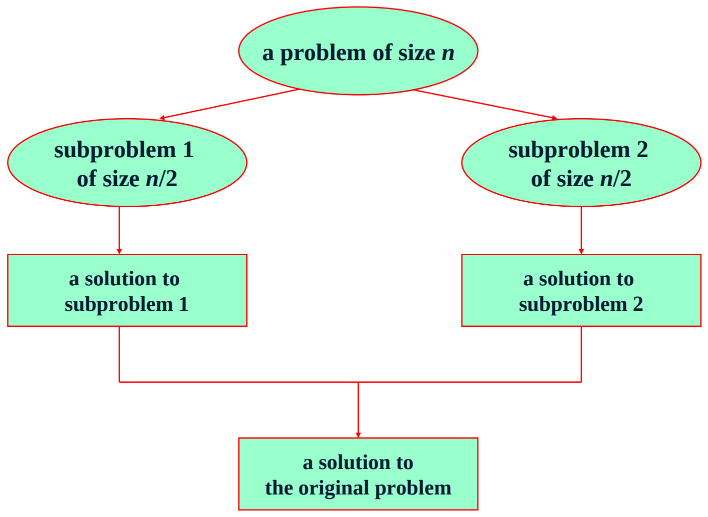
Recurrence Relations : Substitution, Iterative, and The Master Method

Divide and conquer algorithms are common techniques to solve a wide range of problems. They use the following general plan:

1. Divide the problem instance into several smaller instances of the same problem
2. Solve the smaller instances either recursively or directly
3. If necessary, combine the solutions from the smaller instances to get a solution to the original problem

For example, with MergeSort we divided the problem in half, solved each half, then combined the results (merge):


a problem of size *n*
subproblem 1 of size *n*/2
subproblem 2 of size *n*/2
a solution to subproblem 1
a solution to subproblem 2
a solution to the original problem

In this case, the amount of time it takes to run MergeSort (T(n)) is the amount of time it takes to solve the two subproblems (2*T(n/2)) plus the amount of time it takes to split the big problem into the subproblems (constant time) and combine the subproblem solutions back into the solution (O(n) time for merge).

This type of recursive formulation for the runtime is called a recurrence relation. Earlier we said that the recurrence relation for MergeSort is:

Recurrence: $$T(n) = \left\{ \begin{array}{c} \Theta(1), n = 1 \\ 2T\left(\frac{n}{2}\right) + \Theta(n), n > 1 \end{array} \right\}$$

We claimed the runtime is then θ(nlgn).

The next section will show how to compute these bounds. There are generally three methods we will discuss to solve recurrence relations:

Induction
Iterative Substitution
Master Theorem

Note that we will typically drop out the lower order terms.
For example, if an algorithm runs in θ(n) + θ(1) time, the θ(n) will dominate so we can just drop the θ(1) term completely.

Induction Method

The Induction method consists of the following steps:

1. Guess the form of the solution
2. Use mathematical induction to find the constants of the solution, assume the solution works for up to n. We need to then show the solution works for n+1; this is a form of proof by induction.
3. Solve the base case
4. This method works best if its easy to guess the form of the answer

Example: Given

T(n)= K , i.e. θ(1) for n<=1 ; Where K is a constant often not specified and left implicit
T(n)=2T(n/2) + n for n>1 ; similar to what we had for merge sort

Let's guess the form of the solution is O(nlgn).

Next we need to prove that $T(n) \le c\, n \lg(n)$ for c>0 (recall definition of O(nlgn)

This means that $T(n/2) \le c\,(n/2) \lg(n/2)$

Substitute back in to original guess:

$T(n) \le 2(c(n/2)\lg(n/2)) + n$
$T(n) \le cn \lg(n/2) + n$
$T(n) \le cn \lg(n) - cn\lg 2 + n$
$T(n) \le cn \lg(n) - cn + n \quad \le cn \lg n$; TRUE if $c \ge 1$

Looks good so far, we must also show the boundary condition (or base case) fits our guess by showing that T(constant) from the original recurrence $\le$ guess condition.

We should start at T(2) since the recurrence T(n)=2T(n/2)+n only holds for n>1: (Remember for Big-O we only need to show that the solution is true for $n > n_0$ and can ignore transients early on, so it's ok if T(n) doesn't hold for T(1) for example.)

Try $T(2) \le c2lg2$
$T(2) \le c2$

Orig: $T(2) = 2T(n/2)+2$
$T(2) = 2T(1) + 2$
$T(2) = 2K + 2$

Is $2K+2 \le c2$? The guess is true for $c \ge K+1$ so we are done.

In the above example, it is common to "cheat" a little bit and instead assume that $T(1) = 1$ instead of $T(1) = \theta(1)$.

The reason is to simplify the analysis a bit. Let's look at what we would have if we used $T(1)=1$ instead and tried to solve for the base case:

Try $T(2) \le c2$
Orig: $T(2) = 2T(n/2)+2$
$T(2) = 2T(1) + 2$
$T(2) = 2(1) + 2$
$T(2) = 4$

Is $4 \le c2$? Yes, if $2 \le c$. We can still make the base case work for $T(1)= 1$. The difference is that there is a constant factor using the $T(1)=K$, that is missing when we assume $T(1)=1$. For a slightly easier analysis, we will often assume that $T(1)=1$ to simplify the process. However, remember that even if $T(1) = \theta(1)$, it is safe to assume $T(1)=1$ to prove the recurrence.

How to guess:

1. It is often useful to drop the asymptotically insignificant constants.

Ex: $T(n) = 2T(n/2 + 17) + n$

Ignore the constants, since for a large n they are insignificant. Guess $O(nlgn)$

Or, prove a loose upper and lower bounds, then try to reduce the bounds.
E.g., start with $O(n^3)$, $O(n^2)$, … assume only $\Omega(n)$ initially in recurrence.

2. Subtract Terms:

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

If you guess $T(n) \le cn$ linear
Get $T(n) \le cn/2 + cn/2 + 1$
$T(n) \le cn + 1$
This is not $\le$ cn for any value of n!

Instead try subtracting out the lower order term that is causing problems:

Try $T(n) \le cn - b$; $b >= 0$
Get $T(n) \le (cn/2)-b + (cn/2)-b + 1$
$T(n) \le cn - 2b + 1$
$T(n) \le cn - b$; if $b>=1$ then this works

3. Changing Variables:

Consider $T(n) = 2T(\sqrt{n}) + \lg n$; looks hard!

Let $m=\lg n$; $n = 2^m$
Get $T(2^m) = 2T(n^{1/2}) + m$
$T(2^m) = 2T(2^{m/2}) + m$
Let $S(m) = T(2^m)$

$\rightarrow S(m)=2S(m/2) + m$; We solved this already!

$= O(m \lg m) = O(\lg n \lg\lg n)$; This grows very slowly!

Second induction example:

Given : $T(n) = \theta(1)$ if $n=1$
$T(n) = 4T(n/2) + n$ if $n>1$

Let's guess that the solution is $O(n \lg n)$
So we need to show that $T(n) \le cn \lg n$

Show: $T(n) \le 4 (c(n/2)\lg(n/2)) + n$
$T(n) \le 2cn \lg(n) - 2cn \lg 2 + n$
$T(n) \le 2cn \lg n - 2cn + n$
$T(n) \le 2cn \lg n - (2c+1) n$
This is NOT of the form cn lgn for any value of c! Must match the exact form of the induction hypothesis or the result is not valid. Additionally, this result is not $\le$ cnlgn for all $n \ge n_0$.

New try: Guess solution is $O(n^2)$
So we need to show that $T(n) \le cn^2$

Show: $T(n) \leq 4(c(n/2)^2) + n$

$T(n) \leq 4c(n^2/4) + n$

$T(n) \leq cn^2 + n$

Does not work either! But its close.

Off by a factor of n, so lets try subtracting off that term.

Try: $T(n) \leq cn^2 - n$

$T(n) \leq 4((cn^2/4) - (n/2)) + n$

$T(n) \leq cn^2 - 2n + n$

$T(n) \leq cn^2 - n$

This is a match! c = any value > 0.

We also need to show the base case:

In our guess, $T(2) \leq 4c - 2$

In original, $T(2) = 4T(2/2) + 2$

$T(2) = 4+2$

$T(2) = 6$

So our guess is safe as long as c>=2.

Be careful! The solution must match the EXACT form of the induction hypothesis.

Ex: Given $T(n) = 2T(n/2) + n$

Guess $T(n) \leq cn$

Show $T(n) \leq 2c(n/2) + n$

$T(n) \leq cn + n$

It looks like this works, since we have cn+n which is O(n) and something O(n) is $\leq$cn but it is not the exact same form as our guess. It has an extra n in there that the guess does not. (The ultimate solution may still be O(n) though! But this doesn't prove it. For this problem, a guess of $T(n) \leq n\lg n$ would work).

Iterative Substitution Method: Expand the terms into a summation, and solve algebraically

Example:

T(n)= Theta(1) for n=1
T(n) = 3T(n/4) + n for n>1

$$T\left(\frac{n}{4}\right) = 3T\left(\frac{n}{4*4}\right) + \frac{n}{4}$$

We can plug this back into the original recurrence relation:

$$T(n) = 3\left(3T\left(\frac{n}{16}\right) + \frac{n}{4}\right) + n$$

We can keep on going:

$$T(n) = 3\left(3\left(3T\left(\frac{n}{64}\right) + \frac{n}{16}\right) + \frac{n}{4}\right) + n$$

If we stop at this point and do some math:

T(n) = 27T(n/64) + 9(n/16) + 3(n/4) + n

$$T(n) = n + \frac{3n}{4} + \frac{9n}{16} + 27T\left(\frac{n}{64}\right)$$

There’s a pattern here! If we consider i as the index, where i=1 gives us n+(3/4)n, then we can generalize this as i increases:

$$T(n) = n + \frac{3n}{4} + \frac{9n}{16} + \ldots + \frac{3^j n}{4^j} + \ldots + 3^i T\left(\frac{n}{4^i}\right)$$

How far does i go? Does it increase to infinity? NO at some point we will have to stop. But we already know when we stop – we stop at T(1) because at this point there is no more recursion, we just return a constant number for the amount of work to do.

If we stop at T(1), this means we will stop when 1=(n/4^i).

$$1 = \frac{n}{4^i} \qquad n = 4^i \qquad \log_4 n = i$$

So we can now express the recurrence relation as:

$$T(n) = n + \frac{3n}{4} + \frac{9n}{16} + \ldots + \left(\frac{3}{4}\right)^i n + \ldots 3^{\log_4 n}\Theta(1)$$

substituting $\Theta(1)$ for T(n/4^i) since we will only do a constant amount of work on the last iteration.

We can summarize this as a single summation. First recall that

$3^{\log_4 n} = n^{\log_4 3}$; this is sublinear since $\log_4 3 < 1$

$$T(n) = \left(\sum_{i=0}^{\log_4 n - 1} n\left(\frac{3}{4}\right)^i\right) + \Theta(n^{\log_4 3})$$

$T(n) \le \left(n\sum_{i=0}^{\infty}\left(\frac{3}{4}\right)^i\right) + \Theta(n^{\log_4 3})$; up to infinity bigger, so <= applies

recall that $\sum_{k=0}^{\infty} x^k = \frac{1}{1-x}$; for x<1

$$T(n) \le n\frac{1}{1-\frac{3}{4}} + \Theta(n^{\log_4 3})$$

$T(n) \le 4n + \Theta(n^{\log_4 3})$; $T(n) \le 4n + o(n)$; loose upper bound so use little-o

This means that the recurrence is Θ(n).

This method is accurate but can result in a lot of algebra to keep track of; can also get very challenging for more complicated recurrence relations.

Second Example:	T(n)=1	if n=1
	T(n)=4T(n/2)+n	if n>1

T(n) =4T(n/2) + n
=4(4T(n/4)+n/2)+n
=4(4(4T(n/8)+n/4)+n/2)+n
=64T(n/8) + 4n +2n +n
=n + 2n +4n + 64T(n/8)
=n + 2n + 4n + … +2^jn + … 4^iT(n/2^i) ; hard part to figure out

What is the last term? When (n/2^i)=1 → i=lgn

T(n) = n + 2n + 4n + 8n + … 2^in + … $4^{\lg n}$ Θ(1)

$$= \left(n \sum_{i=0}^{\lg n-1} 2^i \right) + 4^{\lg n}\Theta(1)$$

We know that $\sum_{k=0}^{m} x^k = \frac{x^{m+1}-1}{x-1}$

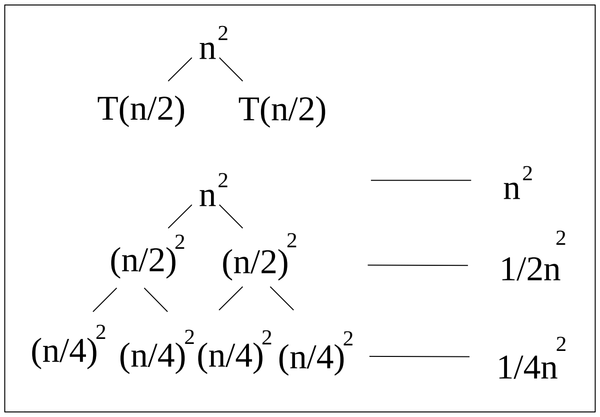
Let's let m=lgn-1. Then:

$$\begin{aligned} T(n) &= n\left(\frac{2^{\lg n-1+1}-1}{2-1}\right)+4^{\lg n}\Theta(1) \\ &= n2^{\lg n}-n+4^{\lg n}\Theta(1) \\ &= n^2 - n + n^{\lg 4}\Theta(1) \\ &= n^2\,(\Theta(1)+1) - n \qquad = \qquad \Theta(n^2) \end{aligned}$$

Sometimes a recursion tree can help:

Recursion Tree: Help to keep track of the iterations

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



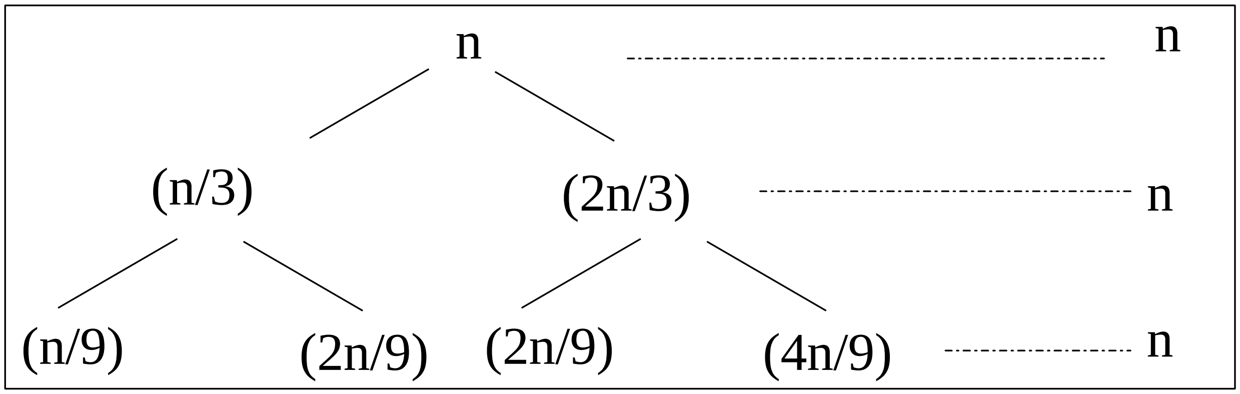
How deep does the tree go?
We stop at the leaf, and we know we're at a leaf when we have a problem of size 1.

$1=(n/2^i)^2$
so $n^2=2^{2i}$; $n=2^i$; $i=\lg n$

The amount of work done is then:

$\sum_{i=0}^{\lg n} \left(\frac{n}{2^i}\right)^2 = \Theta(n^2)$; this is geometrically decreasing in size, so it won't get any bigger than n^2.

One more example: T(n) = T(n/3) + T(2n/3) + n



Each level does work of size n; if we just know the height of the tree, i, the total work is ni.

The tree stops when the leaf is of size 1. The hard part is to figure out the formula based on the height:

$n\left(\frac{2}{3}\right)^i = 1$ (why pick the 2/3 branch and not 1/3?)

$n = \frac{1}{\left(\frac{2}{3}\right)^i} = \left(\frac{3}{2}\right)^i$

$i = \log_{3/2} n$

So the total work is $(\log_{3/2} n)n$ or O(nlog $_{3/2}$ n).

Master Theorem Method

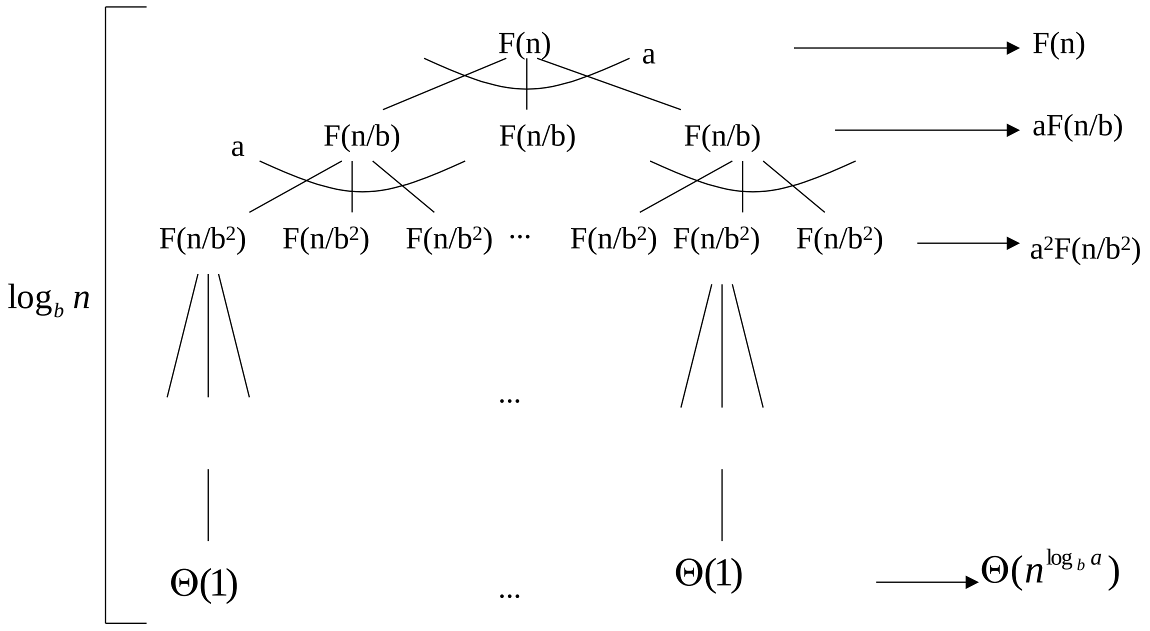
If the form of a recurrence is: $T(n) = aT\left(\frac{n}{b}\right) + f(n), a \geq 1, b > 1$

then we can use the Master Method, which is a cookbook-style method for proving the runtime of recurrence relations that fit its parameters. Note that not all recurrence of the above form can be solved through the master method. We won't prove the master method, but will give an argument as to how it works.

In the master method:

- *a* is the number of subproblems that are solved recursively; i.e. the number of recursive calls.
- *b* is the size of each subproblem relative to n; n/b is the size of the input to the recursive call.
- *f(n)* is the cost of dividing and recombining the subproblems.

Recursion tree example: T(n)=aT(n/b)+f(n)



$$Total = \Theta\left(n^{\log_b a}\right) + \sum_{j=0}^{\log_b n - 1} a^j f\left(\frac{n}{b^j}\right)$$

What is the height of the tree? When $f\left(\frac{n}{b^i}\right) = f(1) \rightarrow \frac{n}{b^i} = 1 \rightarrow n = b^i \rightarrow i = \log_b n$

How many leaves are there?

$$a^{height} = NumberLeaves$$
$$a^{\log_b n} = n^{\log_b a}$$

Work at the leaves is : $\Theta(1)n^{\log_b a} = \Theta\left(n^{\log_b a}\right)$

Work of dividing and combining is: $f(n) + af(\frac{n}{b}) + a^2 f(\frac{n}{b^2}) + \ldots$

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

this does not include the cost of the leaves.

The total work/runtime T(n) is: $\Theta(n^{\log_b a}) + \sum_{i=0}^{\log_b n-1} a^i f(\frac{n}{b^i})$

The time T(n) might be dominated by:

1. The cost of the leaves
2. The cost of the divide/combine of the root
3. Evenly distributed at all the levels

The master method tells us what the asymptotic running time will be depending on which cost is the highest (dominates).

If the form is:

$$T(n) = aT\left(\frac{n}{b}\right) + f(n), a \geq 1, b > 1$$

Then based on comparing *f(n)* and $n^{\log_b a}$ we know the running time given the following three cases:

- If $f(n) = O(n^{\log_b a-\varepsilon})$ for some constant $\varepsilon > 0$ then $T(n) = \Theta(n^{\log_b a})$; cost of leaves dominates.
- If $f(n) = \Theta(n^{\log_b a})$ then $T(n) = \Theta(n^{\log_b a} \lg n)$; cost is evenly distributed
- If $f(n) = \Omega(n^{\log_b a+\varepsilon})$ for some constant $\varepsilon > 0$ and if $af(\frac{n}{b}) \leq cf(n)$ for some constant c<1 and all sufficiently large n, then $T(n) = \Theta(f(n))$; divide/conquer or root cost dominates

Example:

$$T(n) = 9T(\frac{n}{3}) + n$$

So a=9, b=3, f(n)=n

Case 1 works for $f(n) = O(n^{\log_b a-\varepsilon})$. We need to prove this relationship by showing that:

$f(n) = O(n^{\log_b a - \varepsilon})$

$n = O(n^{\log_3 9 - \varepsilon}) = O(n^{2-\varepsilon})$

if $\varepsilon = 1$ then n=O(n) and case 1 is satisfied.

Therefore:

$T(n) = \Theta(n^{\log_b a}) = \Theta(n^{\log_3 9}) = \Theta(n^2)$

In this example, the cost of the leaves has dominated the runtime.

Example:

$T(n) = 2T(\frac{n}{2}) + n$; Merge Sort

So a=2, b=2, f(n)=n

Check case 1:

Is $f(n) = O(n^{\log_b a - \varepsilon})$?

$n = O(n^{\log_2 2 - \varepsilon})$

$n = O(n^{1-\varepsilon})$

For any epsilon>0, n is bigger, so case 1 does not work.

Check case 2:

Is $f(n) = \Theta(n^{\log_b a})$

$n = \Theta(n^{\log_2 2}) = \Theta(n)$ YES

therefore:

$T(n) = \Theta(n^{\log_b a} \lg n) = \Theta(n^{\log_2 2} \lg n) = \Theta(n \lg n)$

Cost is evenly distributed among leaves and upper part of tree.

Example:

$T(n) = T(\frac{2n}{3}) + 1$

So a=1, b=3/2, f(n)=1

Case 1 does not work (exercise for the reader)

Case 2:

Is $f(n) = \Theta(n^{\log_b a})$?

$1 = \Theta(n^{\log_{3/2} 1}) = \Theta(n^0) = \Theta(1)$ YES

therefore:

$T(n) = \Theta(n^{\log_b a} \lg n) = \Theta(n^{\log_{3/2} 1} \lg n) = \Theta(n^0 \lg n) = \Theta(\lg n)$

Cost is again evenly distributed.

Example:

$$T(n) = 3T(\frac{n}{4}) + n\lg n$$

a=3,b=4,f(n)=nlgn

Case 1 and 2 don't fit (exercise for the reader)
Case 3:

Is $f(n) = \Omega(n^{\log_b a+\varepsilon})$?

$n\lg n = \Omega(n^{\log_4 3+\varepsilon}) = \Omega(n^{0.79+\varepsilon})$

YES, if epsilon =0.21, then $n\lg n = \Omega(n)$

We also need to show the extra condition:

Is $af(\frac{n}{b}) \le cf(n)$ for c<1?

$$3f(\frac{n}{4}) \le cf(n)$$

$$3\frac{n}{4}\lg\left(\frac{n}{4}\right) \le cn\lg n$$

$$3\frac{n}{4}(\lg n - \lg 4) \le cn\lg n$$

$$3\frac{n}{4}(\lg n - 2) \le cn\lg n$$

YES, if c=¾ then $3\frac{n}{4}(\lg n - 2) \le \frac{3}{4}n\lg n$

therefore:

$$T(n) = \Theta(f(n)) = \Theta(n\lg n)$$

Example:

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

So a=4, b=2, f(n)= $\frac{n^2}{\lg n}$

Try case 1:

Is $f(n) = O(n^{\log_b a-\varepsilon})$?

$$\frac{n^2}{\lg n} = O(n^{\log_2 4-\varepsilon})$$

$$\frac{n^2}{\lg n} = O(n^{2-\varepsilon})$$

NO, for epsilon>0, f(n) is larger.

Try case 2:

Is $f(n) = \Theta(n^{\log_b a})$?

$$\frac{n^2}{\lg n} = \Theta(n^{\log_2 4}) = \Theta(n^2)$$

NO, grows smaller than n^2.

Try case 3:

Is $f(n) = \Omega(n^{\log_b a+\varepsilon})$?

$$\frac{n^2}{\lg n} = \Omega(n^{2+\varepsilon})$$

NO, for epsilon > 0, f(n) is smaller, not bigger.

Master method does not work for this recurrence relation!
(Solution is $\Theta(n^2 \lg\lg n)$ by substitution)