Recurrence Equations

Recurrence Relation

- A recurrence relation is an equation which is defined in terms of itself
- Many natural functions expressed as recurrences
  - $a_n = a_{n-1} + 1,\ a_1 = 1 \rightarrow a_n = n$ (polynomial)
  - $A_n = 2 * a_{n-1} + 1,\ a_1 = 1 \rightarrow a_n = 2^{n-1}$ (exponential)
- E.g nth Fibonacci number

```
int fib(int n){
  if (n <= 1) return n;
  return fib(n-1) + fib(n-2);
}
```

Recurrences

- Characterize the running times of divide and conquer algorithms
  - Equation or inequality *T*(*n*) that describes a function in terms of its value on smaller inputs
  - Recurrence Equation for Merge Sort

$$t(n)=\begin{cases} b \text{ if } n=1 \\ t(n/2)+t(n/2)+cn \text{ otherwise} \end{cases} \qquad t(n)=\begin{cases} b \text{ if } n=1 \\ 2t(n/2)+cn \text{ otherwise} \end{cases}$$

  - Recurrence for nth Fibonacci
    - T(n) = T(n-1)+ T(n-2) + c

Recurrence Relation

- Recurrence procedure must have a base case
  - Must be small enough
- Can have different forms
  - A recursive algorithm might divide subproblems into unequal sizes, such as a 2/3-to-1/3 split
  - If combine step takes linear time
    - $T(n) = T(2n/3)/ + T(n/3) + \Theta(n)$
  - Can also have subproblems containing only one element fewer than the original problem

$$T(n) = \begin{cases} 3 \; if n = 1 \\ T(n-1) + 7 \; otherwise \end{cases}$$

Solving Recurrences

- Methods for solving recurrences (to obtain asymptotic bounds on the solution)
  - Substitution method
    - guess a bound and then use mathematical induction to prove our guess correct
  - Recursion-tree method
    - convert the recurrence into a tree whose nodes represent the costs incurred at various levels of the recursion
  - Master method
    - Solves equations of the form $T(n)= aT(n/b)+ f(n)$
      - where $a >= 1$, $b > 1$, and $f(n)$ is a function representing the cost of merge

Substitution Method

- Guess the form of the solution
  - Heuristics can be used to guess the solution
  - If recurrence similar to one already seen, guess a similar solution
    - Smaller constants do not affect the solution
  - Prove loose upper and lower bounds on the recurrence and reduce the range of uncertainty
- Use mathematical induction to find constants and show that the solution is correct

Issues

- Might not be clear what the general pattern might look like
  - Guessing solution is not straightforward
  - Must consider the boundary conditions
- Must justify general form using induction

Example

- Example $T(n) = 2T(\lfloor n/2 \rfloor)+n$
  - Guess T(n) = O($n\log n$)
  - Prove that T(n) <= $cn\log n$ for some constant c>0
  - $T(\lfloor n/2 \rfloor) \le c(\lfloor n/2 \rfloor)\log(\lfloor n/2 \rfloor)$
  - $T(n) \le 2(c(\lfloor n/2 \rfloor)\log(\lfloor n/2 \rfloor))+n$

$$\le cn\log(n/2) + n$$

$$= cn\log n - cn\log 2 + n$$

$$= cn\log n - cn + n$$

$$\le cn\log n$$

Substitution Method

- Recurrence Equation for Merge Sort

$$t(n)=\left\{\begin{array}{c} b \text{ if } n=1 \\ t(n/2)+t(n/2)+cn \text{ otherwise} \end{array}\right\}$$

$$t(n)=\left\{\begin{array}{c} b \text{ if } n=1 \\ 2\mathrm{t}(n/2)+cn \text{ otherwise} \end{array}\right\}$$

- Recursively apply e.g t = $2(2t(n/2^2)+(cn/2))+cn$
  - $t(n) = 2^i t(n/2^i) + cn..+cn$
  - This stops when i = log n or $n = 2^i$
  - Hence t(n) = nt(1) + cnlogn = nb + cnlogn

Changing variables

- Example $T(n) = 2T(\lfloor n^{1/2} \rfloor)+\log n$
  - Rename $m = \log n$ *or* $n = 2^m$
  - $T(2^m) = 2T(\lfloor 2^{m/2} \rfloor)+m$
  - Let $S(m) = T(2^m)$, *and equation becomes*
    - $S(m) = 2S(m/2)+m$, *whose solution is* $O(m \log m)$
  - Changing back from $S(m)$ to $T(n)$
    - $T(n) = T(2^m) = S(m)= O(m \log m) = O(\log n \log\log n)$

Sample Problems

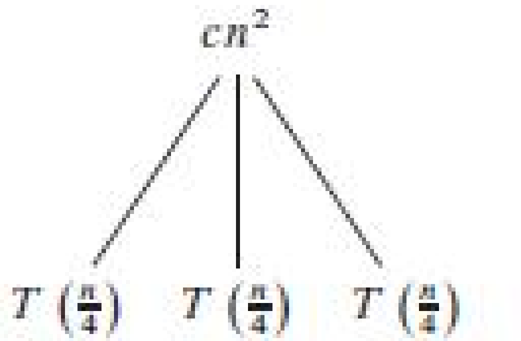
- Show that the solution of $T(n) = T(n-1) + n$ is $O(n^2)$.
- Show that the solution of $T(n) = T(\lceil n/2 \rceil) + 1$ is $O(\lg n)$.
- Solve the recurrence $T(n) = T(\sqrt{n}) + \log n$ by making a change of variables. Your solution should be asymptotically tight.

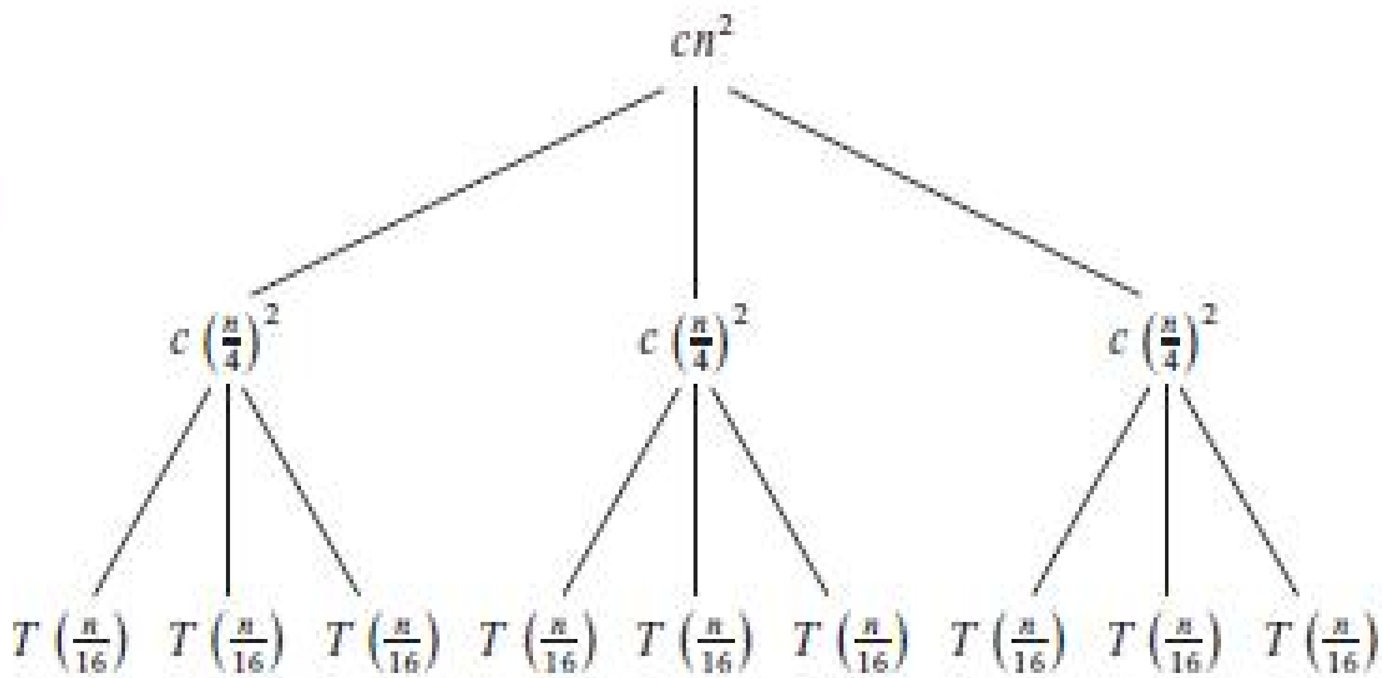
Recursion Tree Method

- Recursion Tree
  - each node represents the cost of a single subproblem somewhere in the set of recursive function invocations
  - Per-level cost computed by summing the node costs within each level
  - Total cost calculated by summing per-level costs over the height of the tree
- Results in a good guess which can be verified by substitution method

Recursion Tree Construction

- $T(n) = 3T(n/4) + cn^2$

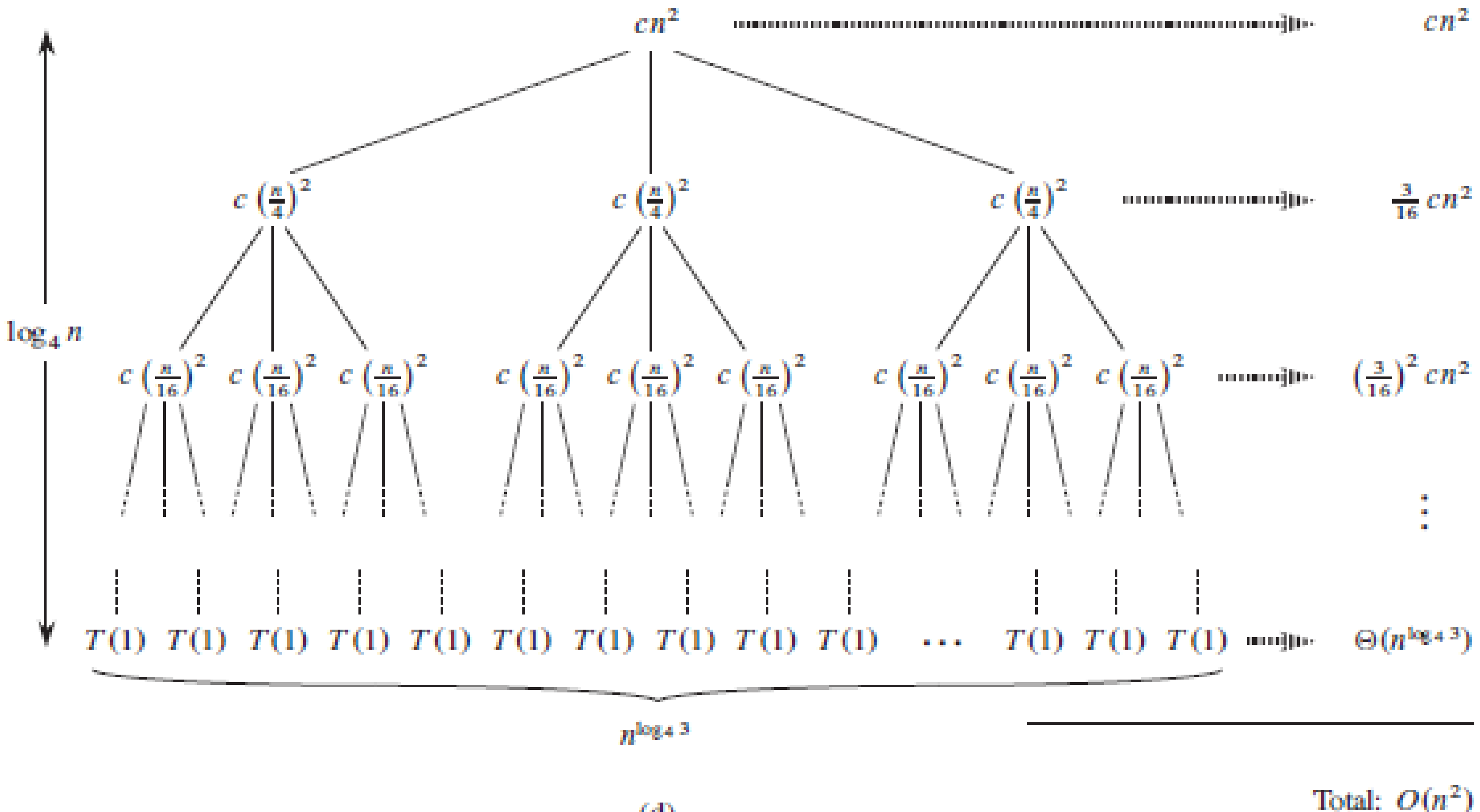




(c)

Src: CLR:Introduction to Algorithms: 4.4

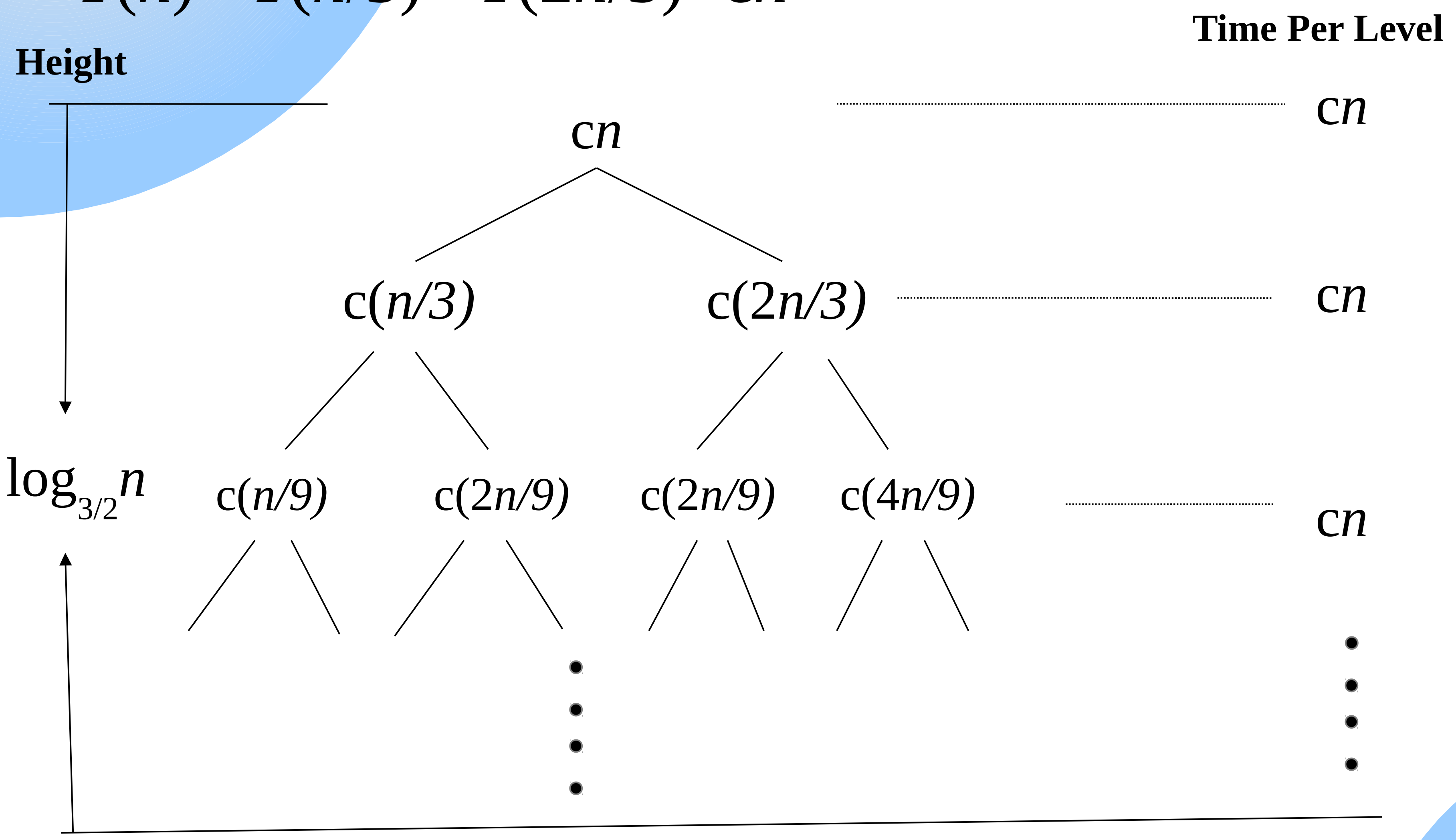
Recursion Tree construction



(d)

Recursion Tree Example2

- $T(n) = T(n/3) + T(2n/3) + cn$



Total Time: $O(n\log n)$

Sample Problems

- Use the recursion tree to determine a good upper bound on the recurrence $T(n) = 3T(\lfloor n/2 \rfloor)+n$. Use substitution method to verify your answer.
- Use the recursion tree to determine a good upper bound on the recurrence $T(n) = 2\mathrm{T}(n\text{-}1)+1$. Use substitution method to verify your answer.
- Show that the solution to $\mathrm{T(n)} = (2\mathrm{T}(\lfloor n/2 \rfloor) + 17)+ \mathrm{n}$ is $\mathrm{O(nlgn)}$.

Master Method

- Used for recurrence equation of the form

$$t(n) = \begin{cases} c \ if \ n < d \\ aT(n/b) + f(n) if \ n \geq d \end{cases}$$

  - where d>=1 is integer constant, a>0, c>0, b>1 real constants
  - f(n) is positive for n>=d
  - subproblems of size n/b, merge time $O(n^d)$ or O(f(n))
- This form arises in divide-conquer algorithms
  - Divides problem into a subproblems of size at most n/b each
  - Solves each recursively and merges the solutions

The Master theorem

- If there is a small constant $\varepsilon > 0$, s.t, f(n) is $O(n^{\log_b a-\varepsilon})$, then T(n) is $\Theta(n^{\log_b a})$
  - f(n) polynomially smaller than special function $n^{\log_b a}$
- If there is a constant $k >= 0$, s.t, f(n) is $O(n^{\log_b a}\log^k n)$, then T(n) is $\Theta(n^{\log_b a}\log^{k+1} n)$
  - f(n) is asymptotically close to the special function
- If there are small constants $\varepsilon > 0$, and $\delta < 1$, s.t, f(n) is $\Omega(n^{\log_b a+\varepsilon})$, and $af(n/b) <= \delta f(n)$, for n>= d then T(n) is $\Theta(f(n))$
  - f(n) polynomially larger than special function

Master Theorem

- Compare function $n^{\log_b a}$ and $f(n)$, to find out which of them are larger
  - Case 1: $n^{\log_b a}$ is larger hence $T(n)= \Theta(n^{\log_b a})$
  - Case 3: $f(n)$ is larger, hence $T(n)= \Theta(f(n))$
  - Case 2: the two functions are the same size, we multiply by a logarithmic factor, and the solution is $T(n)= \Theta(n^{\log_b a} \log n) = \Theta(f(n) \log n)$

Master Method : Case 1

- If there is a small constant $\varepsilon>0$ s.t, f(n) is $O(n^{\log_b a-\varepsilon})$, then $T(n)= \Theta(n^{\log_b a})$
- Example
  - $T(n) = 4T(n/2)+n$
  - $n^{\log_b a}= n^{\log_2 4} = n^2$
  - $f(n)= O(n^{2-\varepsilon})$ for $\varepsilon=1$
  - $T(n)= \Theta(n^2)$ by Master Method

Master Method: Case 2

- If there is a small constant k>=0, s.t, f(n) is $\Theta(n^{\log_b a}\log^k n)$, then $T(n) = \Theta(n^{\log_b a}\log^{k+1} n)$
- Example
  - $T(n) = 2T(n/2)+n\log n$
  - $n^{\log_b a} = n^{\log_2 2} = n$, hence k = 1 for $f(n) = \Theta(n\log n)$
    - Falls in case 2
  - $T(n) = \Theta(n \log^2 n)$ by Master Method

Master Method: Case 3

- If there is a small constant $\varepsilon>0$ and $\delta<1$, s.t, $f(n)$ is $\mathbf{\Omega}(n^{\log_b a+\varepsilon})$, and $af(n/b) <= \delta f(n)$ for $n>= \delta$, then $T(n)= \mathbf{\Theta}(f(n))$
- Example
  - $T(n) = T(n/3)+n$
  - $n^{\log_b a}= n^{\log_3 1} = n^0 = 1$, hence case 3
  - $f(n)$ is $\mathbf{\Omega}(n^{0+\varepsilon})$, for $\varepsilon=1$, and $af(n/b) = n/3 = (1/3)f(n)$
  - $T(n)= \mathbf{\Theta}(n)$ by Master Method

Problems

- Characterize the following recurrence equations using the master method (assuming T(n) = c for n <d, for constants c >0 and d ≥ 1
  - $T(n) = 2T(n/2) + \log n$
  - $T(n) = 8T(n/2) + n^2$
  - $T(n) = 2T(n/4) + n$
  - $T(n) = 2T(n/4) + n^2$
  - $T(n) = 2T(n/4) + n^{1/2}$

Problem

- Which of the following algorithms is best and why?
  - Algorithm A solves problems by dividing them into 5 subproblems of half the size, recursively solving each subproblem, and then combining solutions in linear time
  - Algorithm B solves problems of size n by recursively solving subproblems of size n-1 and then combining solutions in constant time
  - Algorithm C solves problems of size n by dividing them into 9 subproblems of size n/3, recursively solving each subproblem, and then combining solutions in $O(n^2)$ time