



Algorithm Design and Analysis

CS240

Spring 2024

Rui Fan



Course info

- Assoc. Prof. Rui Fan / 范睿
 - Email: fanrui@shanghaitech.edu.cn
 - Office: SIST 1A-504E
 - Office hours Thursdays 3-4PM.
 - My research is parallel and distributed computing.
- Lecture notes on Blackboard, discussions on Piazza, grading on GradeScope.
- References
 - *Algorithm Design*. Kleinberg, Tardos.
 - *Introduction to Algorithms, 3rd edition*. Cormen, Leiserson, Rivest, Stein.



Grading

Problem sets	Project	Midterm	Final exam
35%	15%	20%	30%
~6 problem sets	Due end of week 16	Week 7 in class	

■ Recitations

- ☐ Problem set solutions and discussions.
- ☐ TAs, recitation time / place TBA.

■ Project

- ☐ Write programs to solve programming contest style algorithm problems with time and memory limits.

Random number generator Reference counting Randomized algorithms
Euclid's algorithm Markov chain Monte Carlo Recursive functions Power iteration
Binary search Polynomial time Ackermann function Network flow
Quantum algorithms Fast Fourier Transform Public key encryption Reed-Solomon codes
Chinese remainder theorem Huffman encoding Linear programming
NP-completeness Sieve of Eratosthenes Graph isomorphism
Deep learning Strassen's algorithm Painter's algorithm Preconditioning Auction algorithms
Interior point methods Computational biology Divide and conquer Cuthill-McKee
Traveling salesman problem Zero knowledge proofs Clustering Elliptic curve factorization
Spline interpolation Johnson-Lindenstrauss theorem Principle component analysis
DNA sequence alignment Satisfiability Primal-dual algorithms Newton's method
Voronoi diagram Parallel algorithms List scheduling Quicksort
Convex hulls External memory algorithms Strongly connected components Support vector machines
Branch and bound Dynamic programming Quad trees Secret sharing Maximum matching
Dijkstra's algorithm AES algorithm Bayesian inference Red-black trees Kolmogorov complexity
Splay trees LRU caching Lamport clocks Integer programming Graph coloring
Minimum spanning tree Maximum independent set Mutual exclusion Discrete logarithms
Streaming algorithms Online algorithms AKS primality test
B-trees MPEG compression Bloom filters Simplex algorithm Davis-Putnam algorithm
Approximation algorithms Union-find Clock synchronization Byzantine agreement
Ellipsoid algorithm Nondeterministic finite automata Conjugate gradient Chaitin's algorithm Topological sort
VC dimension Distributed algorithms Ray tracing N-body problems
Neural networks PageRank Perfect hashing



Course content

- Analysis of algorithms
- Divide and conquer
- Greedy algorithms
- Dynamic programming
- Network flow
- NP and complexity
- Overcoming intractability
- Randomized algorithms
- Approximation algorithms



What is an algorithm?

- A precise, step-by-step procedure for solving a problem.
 - Take an input (an **instance** of the problem).
 - Perform a sequence of **operations** on data from the instance.
 - Produce an output (**solution** to the instance).



Expressing algorithms

- An **algorithm** is a method for solving a given problem.
- A **program** expresses the algorithm in a way a computer can understand.
 - Can use many different languages (C / C++ / Java / Python / ...) to express the same algorithm.
- **Data structures** are different ways to store data used by an algorithm.
 - **Ex** A dictionary can be stored as linked list, array, tree, etc.
 - Using the right data structure makes an algorithm more efficient.

Pseudocode

- We'll mostly write our algorithms in pseudocode.
- Precisely captures main ideas of algorithm without getting bogged down in details.
- You should practice translating between pseudocode and real code.

```
MAX-HEAPIFY(A, i)
1  l = LEFT(i)
2  r = RIGHT(i)
3  if l ≤ A.heap-size and A[l] > A[i]
4      largest = l
5  else largest = i
6  if r ≤ A.heap-size and A[r] > A[largest]
7      largest = r
8  if largest ≠ i
9      exchange A[i] with A[largest]
10     MAX-HEAPIFY(A, largest)
```


Comparing algorithms

- A problem can be solved by many **different algorithms**.
 - Some algorithms are better than others.
- We focus on the **time** and **memory** complexity of an algorithm.
 - The less time and memory an algorithm uses, the better.
 - Other important measures include speed on real hardware, parallelism, energy use, simplicity and elegance, etc.
 - Can also compare amount of randomness needed, approximation ratio, competitive ratio, etc.
- Good algorithms are the key to efficiency.
 - **Ex** Use two algorithms to sort 10M numbers, on a processor which takes 1 billion steps per second.

	Algorithm A	Algorithm B
Complexity	n^2	$n \log_2 n$
Sorting time	$\frac{(10^7)^2}{10^9} \approx 28 \text{ hours}$	$\frac{10^7 * \log_2 10^7}{10^9} \approx 0.024 \text{ seconds}$

- Better algorithms are more important than faster hardware.
 - **Ex** Even if processor speed doubles every year, in 10 years algorithm A would still take ~100 seconds.



Time complexity

- Time complexity of an algorithm is the **number of steps** it performs until it terminates.
- A good complexity measure needs to address several issues.
- **Issue 1** Complexity depends on input.
- **Solution** Analyze complexity as a function of input size.
 - **Ex** Adding two n digit numbers takes n steps.
 - **Ex** Multiplying two n digit numbers takes n^2 steps.
- **Issue 2** For fixed input size, running time can still vary.
- **Solution** For a given input size, consider **worst case**, i.e. maximum possible number of steps.
 - **Ex** Finding item in a size n linked list takes at most n steps.
- Sometimes also consider **average case** complexity, i.e. average number of steps, over all inputs of certain size.
 - But this depends on knowing how likely each input is.
 - An algorithm tuned for one input distribution may perform poorly on another.



Time complexity

- **Issue 3** Number of steps depends on language and hardware details.
 - **Ex** Processor A does one arithmetic operation per step. Processor B does an add and multiply each step.
 - Computing $\vec{x} \cdot \vec{y} = \sum_{i=1}^n x_i y_i$ takes $2n$ steps on processor A and n steps on processor B.
- **Solution Ignore** constant factors in time complexity.
 - **Ex** Count $2n, n, 100n$ and $0.01n$ as the same thing.
 - **Ex** But n^2 and n are different, because they differ by nonconstant factor.
- **Issue 4** Speeds of two algorithms can flip as inputs get larger.
 - **Ex** Algorithm A is faster than algorithm B for small inputs, but slower for big inputs.
- **Solution** Focus on **asymptotic complexity**, i.e. very large inputs.

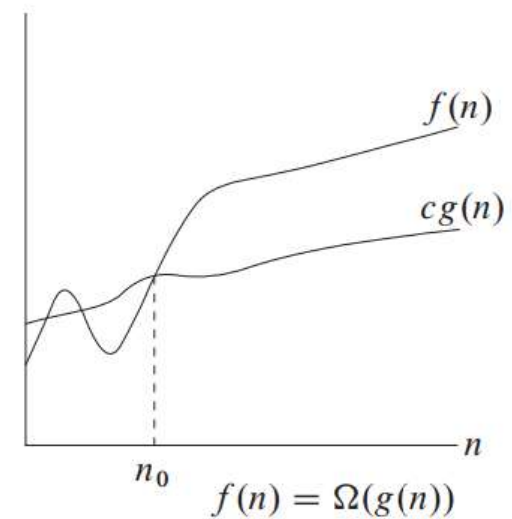
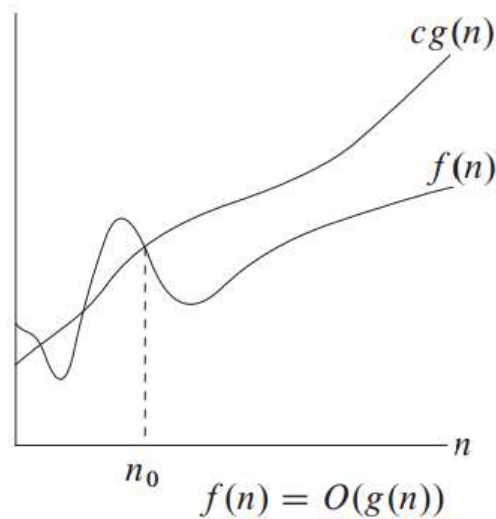
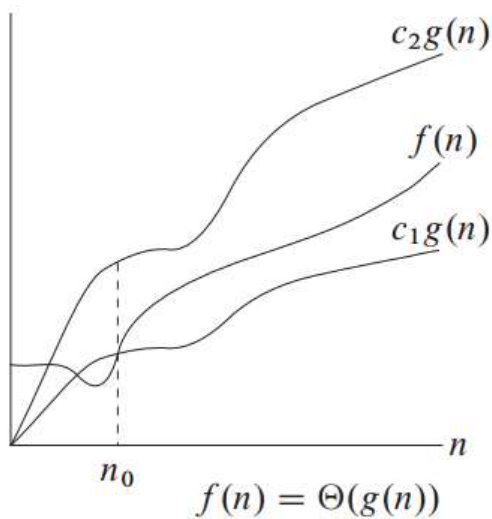


Asymptotic analysis

- Compare sizes of functions $f(n)$ and $g(n)$ when ignoring constant factors and small inputs.
- Sometimes called “big O notation”.

Notation	Intuitive meaning	Formal definition
$f(n) = O(g(n))$	$f \leq g$	$\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} < \infty$
$f(n) = \Omega(g(n))$	$f \geq g$	$\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} > 0$
$f(n) = \Theta(g(n))$	$f = g$	$\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = c, \quad 0 < c < \infty$
$f(n) = o(g(n))$	$f < g$	$\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = 0$
$f(n) = \omega(g(n))$	$f > g$	$\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = \infty$

Big O pictorially



Source: *Introduction to Algorithms*
Cormen, Leiserson, Rivest, Stein



Examples 1

Example	Proof
$n = O(2n)$	$\lim_{n \rightarrow \infty} \frac{n}{2n} = \frac{1}{2} < \infty$
$10n^2 = O(n^2)$	$\lim_{n \rightarrow \infty} \frac{10n^2}{n^2} = 10 < \infty$
$n = O(n^2)$	$\lim_{n \rightarrow \infty} \frac{n}{n^2} = 0 < \infty$
$n^2 = O(2^n)$	$\lim_{n \rightarrow \infty} \frac{n^2}{2^n} = 0 < \infty$
$n \neq O(\sqrt{n})$	$\lim_{n \rightarrow \infty} \frac{n}{\sqrt{n}} = \lim_{n \rightarrow \infty} \sqrt{n} = \infty$



Examples 2

Example	Proof
$n = \Omega(2n)$	$\lim_{n \rightarrow \infty} \frac{n}{2n} = \frac{1}{2} > 0$
$10n^2 = \Omega(n^2)$	$\lim_{n \rightarrow \infty} \frac{10n^2}{n^2} = 10 > 0$
$n^2 = \Omega(n)$	$\lim_{n \rightarrow \infty} \frac{n^2}{n} = \infty > 0$
$n = \Omega(\log n)$	$\lim_{n \rightarrow \infty} \frac{n}{\log n} = \infty > 0$
$n^2 \neq \Omega(2^n)$	$\lim_{n \rightarrow \infty} \frac{n^2}{2^n} = 0$



Examples 3

Example	Proof
$n = \Theta(2n)$	$\lim_{n \rightarrow \infty} \frac{n}{2n} = \frac{1}{2}$
$10n^2 = \Theta(n^2)$	$\lim_{n \rightarrow \infty} \frac{10n^2}{n^2} = 10$
$n^2 \neq \Theta(n)$	$\lim_{n \rightarrow \infty} \frac{n^2}{n} = \infty$
$n^2 \neq \Theta(2^n)$	$\lim_{n \rightarrow \infty} \frac{n^2}{2^n} = 0$

Big O properties

- If $f(n) = O(g(n))$ and $f(n) = \Omega(g(n))$, then $f(n) = \Theta(g(n))$.

- O, Ω, Θ are transitive.

- **Ex** If $f(n) = O(g(n))$ and $g(n) = O(h(n))$, then $f(n) = O(h(n))$.

Proof. Since $f(n) = O(g(n))$, we have $\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = c < \infty$. Also, since $g(n) = O(h(n))$, we have $\lim_{n \rightarrow \infty} \frac{g(n)}{h(n)} = c' < \infty$. Thus, $\lim_{n \rightarrow \infty} \frac{f(n)}{h(n)} = (\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)}) (\lim_{n \rightarrow \infty} \frac{g(n)}{h(n)}) = cc' < \infty$, and so $f(n) = O(h(n))$. \square

- Θ is symmetric.

- I.e. if $f(n) = \Theta(g(n))$, then $g(n) = \Theta(f(n))$.

- $O(1)$ is the set of constants.

- I.e. any $c = O(1)$.



Analyzing complexity

- All programs can be written using **loops**, **if-else** structures, and **recursion**.
- Analyze the complexity of each of type of structure.

Loops

- For and while loops.
 - Loops can be nested.
- Count everything in inner loop as one step.
 - Assume no function calls in inner loop.
 - There's constant number of steps in inner loop, i.e. $O(1)$ steps.
- Ex Complexity is $O(n)$.
- Ex Complexity is $1 + 2 + \dots + (n - 1) = \frac{n(n-1)}{2} = O(n^2)$.
- Ex Complexity is $\lceil \log_2 n \rceil = O(\log n)$.
 - After $\lceil \log_2 n \rceil$ steps, $i = 2^{\lceil \log_2 n \rceil} \geq n$, so the loop terminates.

```
for(i=0; i<n; i++) {  
    j = j+i;  
    k = k*j;  
}
```

```
for(i=0; i<n; i++) {  
    for(j=0; j<i; j++) {  
        printf("*");  
    }  
    printf("\n");  
}
```

```
*  
**  
***  
****  
*****
```

```
i=1;  
while (i<n) {  
    i=i*2;  
    // do stuff  
}
```

if-else statements

- We don't know which branch we'll run.
- Since want worst case complexity, assume the longest branch runs.
- **Ex** Branch 1 does n steps, branch 2 does n^2 steps. Since $n^2 \geq n$, step complexity is n^2 .

```
if (x==1)
    // do stuff A
else if (x==2)
    // do stuff B
...
else
    // do stuff Z
```

```
if (x==1)
    for(i=0;i<n;i++) {
        // do stuff
    }
else if (x==2)
    for(i=0;i<n;i++)
        for(j=0;j<n;j++) {
            // do stuff
        }
```


Recursive functions

- **Recursive** functions can call themselves.
- Many problems are “self reducible”, i.e. we can solve the problem by first solving **smaller instances** of the problem.
- Natural to use recursive algorithm to solve these problems.
- There must be a **base case** that's solvable directly, without using recursion.
- **Ex** Let $\text{sum}(n) = 1 + 2 + \dots + n$.
 - Then $\text{sum}(n) = n + \text{sum}(n-1)$.
 - The base case is $n=1$, for which $\text{sum}(1)=1$.

```
int sum(int n) {  
    if (n==1)  
        return 1;  
    else  
        return n+sum(n-1);  
}
```



Analyzing recursive algorithms

- Two main steps.
 - Find a **recurrence relation** for the time complexity.
 - Solve the recurrence relation.
- Several ways to solve a recurrence relation.
 - Solve it directly, e.g. based on a guess.
 - Substitution method.
 - Recursion tree.
 - Master method.
- For first three methods, need to prove solution is correct using mathematical induction.

Finding a recurrence relation

- Given a function, let $S(n)$ be the (worst case) number of steps it takes on an input of size n .
- The recurrence relation expresses $S(n)$ as a **function of itself**.
 - Also need a base case when n is small.
- **Ex** $S(n) = 1 + S(n - 1)$
 - Base case $S(1) = 1$, since we just do return when $n = 1$.
 - For $n > 1$, we do one step (+), then call $\text{sum}(n-1)$, which takes $S(n - 1)$ steps.
- **Ex** $S(n) = n + S(n - 2)$
 - Base cases $S(0) = S(1) = 1$.
 - For $n > 1$, we do n steps in the for loop. Then we call $\text{foo}(n-2)$, which takes $S(n - 2)$ steps.

```
int sum(int n) {  
    if (n==1)  
        return 1;  
    else  
        return n+sum(n-1);  
}
```

```
void foo(int n) {  
    if (n<=1)  
        return;  
    for(i=0; i<n; i++) {  
        // do stuff }  
    return foo(n-2);  
}
```

Direct solution

- First guess a solution (based on a pattern), then prove it using mathematical induction.
- Ex $S(n) = n + S(n - 2), S(0) = S(1) = 1$.
 - Consider odd $n = 2m - 1$. Even case similar.
 - $S(1) = 1, S(3) = 4, S(5) = 9, S(7) = 16$, etc.
 - So we guess $S(n) = m^2$.
- Prove this by induction.
 - Base case $S(1) = 1 = 1$.
 - Assume we proved it up to $n = 2m - 1$.
 - For next odd n , we have
$$S(n + 2) = S(2(m + 1) - 1) = n + 2 + S(n) = 2m + 1 + m^2 = (m + 1)^2.$$
 - Second equality is the recurrence relation. Third equality is the inductive hypothesis.

```
void foo(int n) {  
    if (n<=1)  
        return;  
    for(i=0; i<n; i++) {  
        // do stuff }  
    return foo(n-2);  
}
```

Substitution method

- First define the recurrence relation.
 - Let $S(n)$ be number of steps $\text{bar}(n)$ takes.
 - Base case $S(1) = 1$.
 - For $n > 1$, $S(n) = 1 + S(n/2)$.
- To solve for $S(n)$, keep substituting the recurrence relation into itself.
- $S(n) = 1 + S(n/2)$.
 - Main recurrence relation.
- $S(n/2) = 1 + S(n/4)$.
 - By substituting $n/2$ into the main relation.
- $S(n/4) = 1 + S(n/8)$.
 - By substituting $n/4$ into the main relation.
- Etc.

```
int bar(int n) {  
    if (n<=1)  
        return 0;  
    return 1+bar(n/2);  
}
```

Substitution method

- Assume first $n = 2^k$ for some k .
- Base case $S(n/2^k) = S(1) = 1$.
- Now do back substitution.

$$\begin{aligned} S(n) &= 1 + S\left(\frac{n}{2}\right) = 1 + 1 + S\left(\frac{n}{4}\right) = 1 + \\ &1 + 1 + S\left(\frac{n}{8}\right) = \dots = 1 + 1 + \dots + 1 + \\ &S(1) = 1 + 1 + \dots + 1. \end{aligned}$$

□ There are $k + 1$ 1's in the final expression,
so $S(n) = k + 1$.

- Since $n = 2^k$, then $k = \log_2 n$, and $S(n) = \log_2 n + 1$.
- General case is similar. Show that $S(n) = \lfloor \log_2 n \rfloor + 1$.

```
int bar(int n) {  
    if (n<=1)  
        return 0;  
    return 1+bar(n/2);  
}
```


Response	Percentage
Yes, the current government is responsible	92%
No, the current government is not responsible	8%

-
- The diagram illustrates the merging of two sorted arrays, $[38, 27, 43, 3]$ and $[9, 82, 10]$, into a final sorted array $[3, 9, 10, 27, 38, 43, 82]$. The process is shown in 10 steps, with each step represented by a box containing the current state of the arrays. Arrows indicate the movement of elements between arrays.
- Step 1: Initial arrays: $[38, 27, 43, 3]$ and $[9, 82, 10]$.
- Step 2: $[38, 27]$ and $[43, 3]$ (from the first array); $[9, 82, 10]$ (from the second array).
- Step 3: $[38]$ and $[27]$ (from the first array); $[43, 3]$ (from the first array); $[9, 82]$ (from the second array); $[10]$ (from the second array).
- Step 4: $[38]$ and $[27]$ (from the first array); $[43]$ and $[3]$ (from the first array); $[9]$ and $[82]$ (from the second array); $[10]$ (from the second array).
- Step 5: $[27, 38]$ (from the first array); $[3, 43]$ (from the first array); $[9, 82]$ (from the second array); $[10]$ (from the second array).
- Step 6: $[3, 27, 38, 43]$ (from the first array); $[9, 10, 82]$ (from the second array).
- Step 7: Final sorted array: $[3, 9, 10, 27, 38, 43, 82]$.

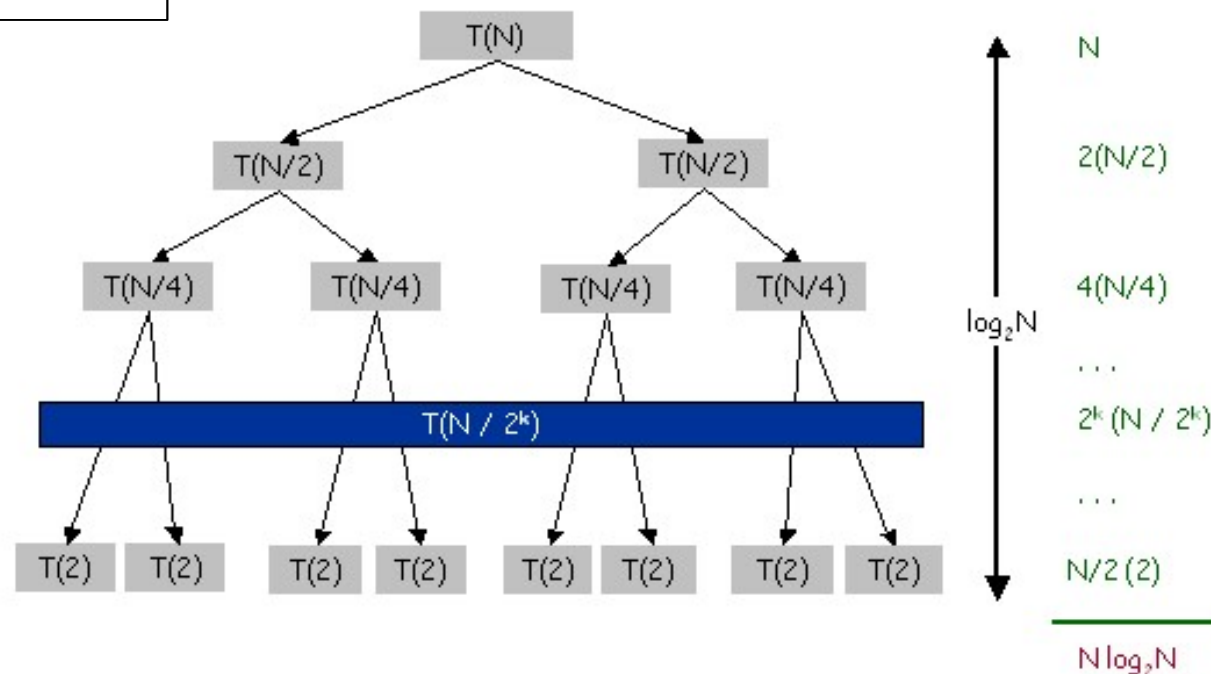
Source: *Wikipedia*

Source: *Wikipedia*

Recursion tree method

```
void mergesort(n) {  
    if (n==1)  
        return;  
    else {  
        L=mergesort(n/2);  
        R=mergesort(n/2);  
    }  
    // takes O(n) time  
    merge(R,L);  
}
```

- ❑ Visualize the recursive calls that occur during mergesort(n).
- ❑ There are $\log_2 n$ levels in the recursion tree.
- ❑ At level i , there are 2^i recursive calls mergesort($n/2^i$).
- ❑ Each call does $n/2^i$ work in merge function.
 - ❑ So total work at level i is n .
- ❑ So total work overall is $S(n) = n \log_2 n$.



Source: <http://www.comscigate.com/cs/IntroSedgewick/40adt/42sort/images/nlogn.png>

Master theorem

- “Plug and play” method for solving a common type of recurrence.
 - Based on comparing the nonrecursive complexity $f(n)$ with $n^{\log_b a}$.

Theorem 4.1 (Master theorem)

Let $a \geq 1$ and $b > 1$ be constants, let $f(n)$ be a function, and let $T(n)$ be defined on the nonnegative integers by the recurrence

$$T(n) = aT(n/b) + f(n),$$

where we interpret n/b to mean either $\lfloor n/b \rfloor$ or $\lceil n/b \rceil$. Then $T(n)$ has the following asymptotic bounds:

1. If $f(n) = O(n^{\log_b a - \epsilon})$ for some constant $\epsilon > 0$, then $T(n) = \Theta(n^{\log_b a})$.
2. If $f(n) = \Theta(n^{\log_b a})$, then $T(n) = \Theta(n^{\log_b a} \lg n)$.
3. If $f(n) = \Omega(n^{\log_b a + \epsilon})$ for some constant $\epsilon > 0$, and if $af(n/b) \leq cf(n)$ for some constant $c < 1$ and all sufficiently large n , then $T(n) = \Theta(f(n))$. ■

Master theorem examples

- **Ex** $T(n) = 9T\left(\frac{n}{3}\right) + n$
 - $a = 9, b = 3, f(n) = n, \log_b a = 2.$
 - Check $f(n) = O(n^{2-\epsilon})$, so use case 1 of Master theorem.
 - So $T(n) = \Theta(n^2).$
- **Ex** $T(n) = T\left(\frac{2n}{3}\right) + 1.$
 - $a = 1, b = \frac{3}{2}, f(n) = 1, \log_b a = 0.$
 - $f(n) = \Theta(n^0)$, so use case 2 of theorem.
 - So $T(n) = n^0 \log n = \Theta(\log n).$
- **Ex** $T(n) = 3T\left(\frac{n}{4}\right) + n \log n.$
 - $a = 3, b = 4, f(n) = n \log n, \log_b a \approx 0.793.$
 - $f(n) = \Omega(n^{0.793+\epsilon})$, so use case 3 of theorem.
 - So $T(n) = \Theta(n \log n).$



Master theorem caveats

- Note in cases 1 and 3, $f(n)$ needs to be smaller (resp. larger) than $n^{\log_b a}$ by a polynomial factor n^ϵ .
 - If this doesn't hold, we can't use the theorem.
- Ex $T(n) = 2T\left(\frac{n}{2}\right) + n \log n$.
 - $a = 2, b = 2, f(n) = n \log n, \log_b a = 1$.
 - However, case 2 of the Master theorem doesn't apply, since $f(n) \neq \Theta(n)$.
 - Case 3 also doesn't apply, since $n \log n \neq \Theta(n^{1+\epsilon})$ for any $\epsilon > 0$.
 - So we can't use the Master theorem to solve this recurrence.
- For a proof of the Master theorem, see Section 4.5 in Cormen et al.
 - Proof basically formalizes the recursion tree method.