Lecture 6: Divide-and-Conquer Analysis Lower Bound on Sorting

Knowledge Has Organizing Power

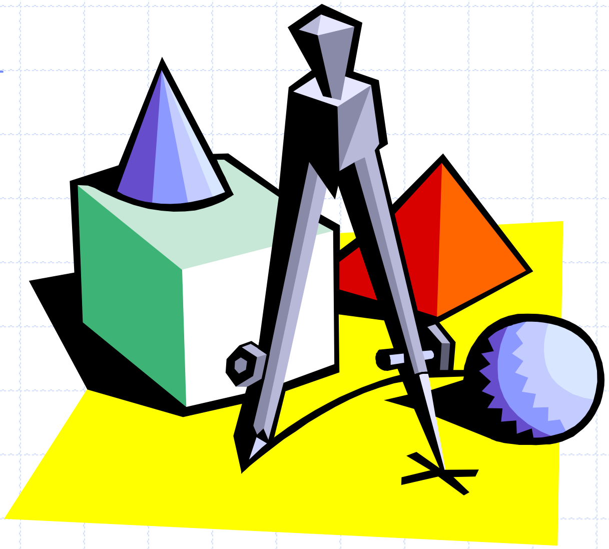
Wholeness Statement

Recursive algorithms are not always easy, in general, to analyze to determine a tight bound on running time. The first step in analyzing divide-and-conquer algorithms is to directly translate them into a recurrence relation that can then be translated directly into a precise tight bound on the algorithm's time complexity. Mathematical techniques make it easy to find a tight bound (e.g., using the Master Theorem). *Science of Consciousness*: It has long been thought that the unified field of pure consciousness is difficult and possibly beyond direct access. The TM technique and scientific research verify that it is easy and possible for the benefit of the individual and society; TM makes it simple and easy for everyone.

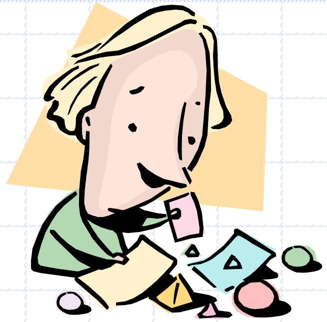
Outline

- Lower Bound on Comparison-Based Sorting (§4.4)
- Divide-and-Conquer Analysis(§5.2)
  - Recurrence Equations (§5.2.1)
  - Master Theorem for solving recurrence equations

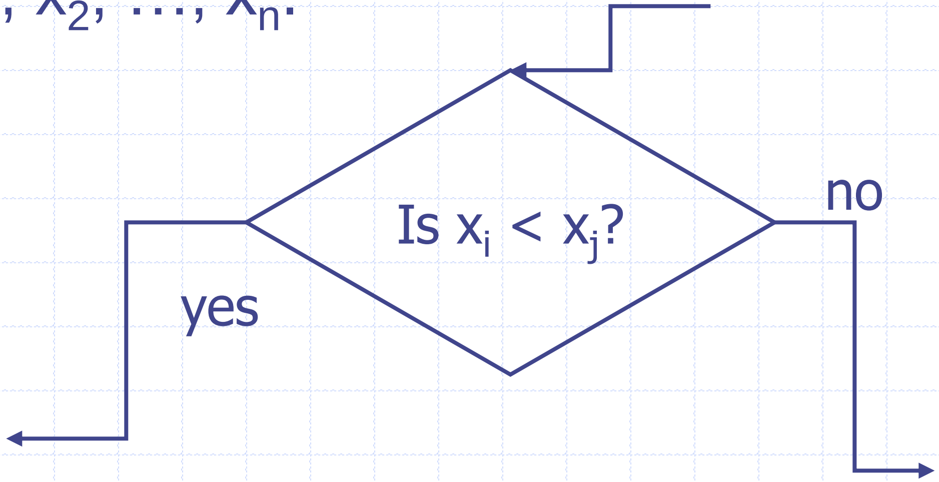
Sorting Lower Bound



Comparison-Based Sorting (§ 4.4)



- Many sorting algorithms are comparison based.
  - They sort by making comparisons between pairs of objects
  - Examples: bubble-sort, selection-sort, insertion-sort, heap-sort, merge-sort, quick-sort, shell sort, ...
- Let us therefore derive a lower bound on the running time of any algorithm that uses comparisons to sort n elements, $x_1, x_2, \ldots, x_n$.

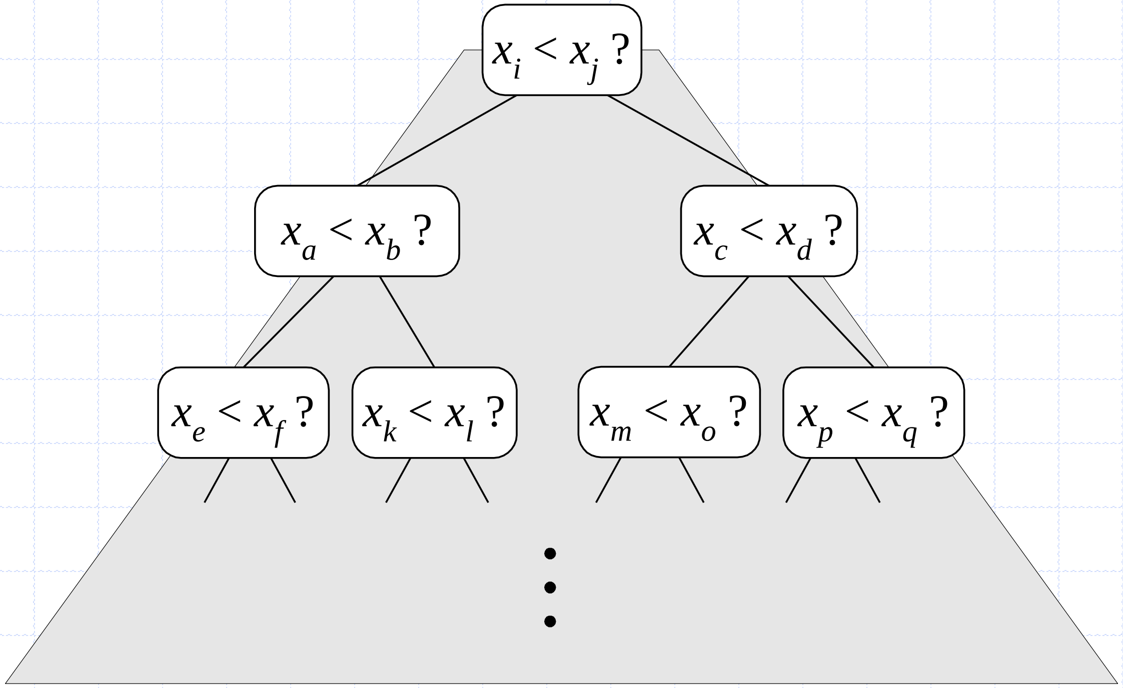


Definition of a Decision Tree

- Internal nodes correspond to key comparisons
  - Thus number of comparisons corresponds to the number of internal nodes
- Leaf nodes correspond to the resulting sorted sequence
- Left subtree shows the next comparison when $x < y$
- Right subtree shows the next comparison when $x \geq y$
- Make the tree as efficient as possible by
  - Removing nodes with single children
  - Removing any paths not followed

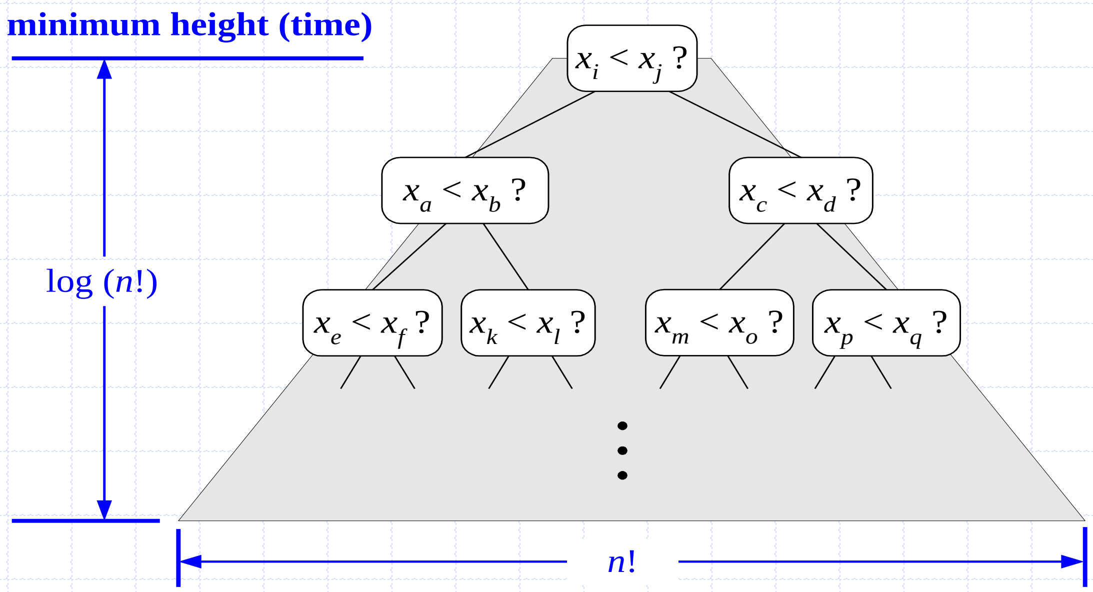
Counting Comparisons

- Each possible run of the algorithm corresponds to a root-to-leaf path in a **decision tree**



Decision Tree Height

- The height of the decision tree is a lower bound on the running time
- Every possible input permutation must lead to a separate leaf output.
  - If not, some input …4…5… would have same output ordering as …5…4…, which would be wrong.
- Since there are n!=1*2*…*n leaves, the height is at least log (n!)



Worst Case Lower Bound

- Number of nodes on the longest path
- i.e., the height h of the decision tree

$2^h \geq n!$ (number of leaf nodes)

$\log 2^h \geq \log n!$

$h \geq \log n!$

$$\begin{aligned} h \geq \log n! &\geq \log (n/2)^{n/2} \\ &= n/2 \log n/2 \\ &= n/2 (\log n - \log 2) \\ &= 1/2 (n \log n - n) \end{aligned}$$

- Thus h is $\Omega(n \log n)$

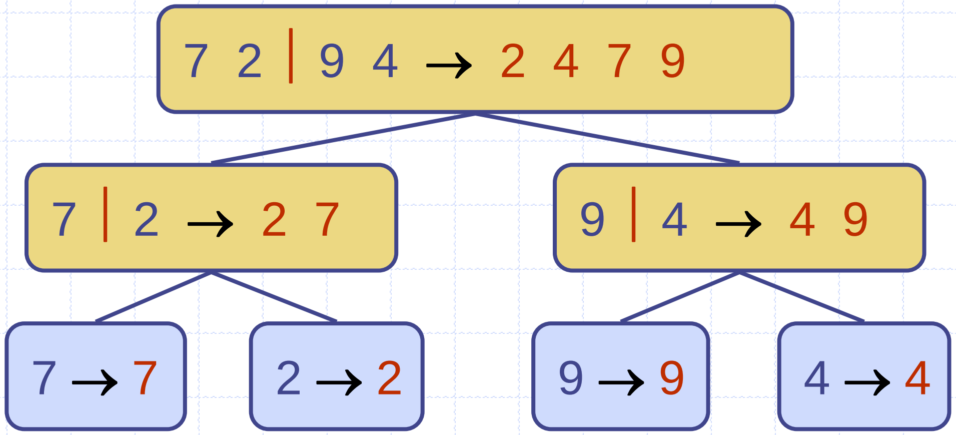
Average Behavior Lower Bound

- Assuming all inputs are equally likely, the average path length can also be shown to be $\Omega(n \log n)$
- Thus the average comparisons to sort by comparison of keys is $\Omega(n \log n)$
- Therefore, no sorting algorithm that sorts by comparison of keys can do substantially better than Heapsort, Quicksort, or Merge-sort

Main Point

1. Any algorithm that sorts n items by comparison of keys must do $\Omega(n \log n)$ comparisons in the worst and average case. Heapsort and Merge-sort come very close to realizing this lower bound; thus $\Omega(n \log n)$ is close to being a maximal lower bound.
   Science of Consciousness: In enlightenment, one realizes the Absolute in the relative for maximal power to fulfill one’s goals.

Divide-and-Conquer Analysis

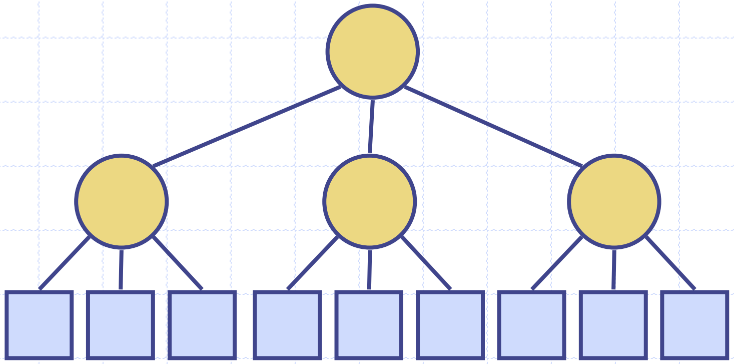


Recurrence Equations

- An equation or inequality that describes a function in terms of its value on smaller inputs
- AKA Recurrence Relations
- Why do we care about solving recurrence equations?
- Four ways for solving
  1) Recursion-tree method
  2) Iterative Substitution method
  3) Guess-and-test method
  4) Master method

Divide-and-Conquer

- Divide-and conquer is a general algorithm design strategy:
  - Divide: divide the input data S in two or more disjoint subsets S_1, S_2, …
  - Recur: solve the subproblems recursively
  - Conquer: combine the solutions for S_1, S_2, …, into a solution for S
- The base case for the recursion are subproblems of constant size
- Analysis can be done using **recurrence equations**



Merge-Sort Revisited

- Merge-sort on an input sequence S with n elements consists of three steps:
  - Divide: partition S into two sequences S_1 and S_2 of about $n/2$ elements each
  - Recur: recursively sort S_1 and S_2
  - Conquer: merge S_1 and S_2 into a unique sorted sequence

```
Algorithm mergeSort(S, C)
    Input sequence S with n
        elements, comparator C
    Output sequence S sorted
        according to C
    if S.size() > 1 then
        (S_1, S_2) ← partition(S, n/2)
        mergeSort(S_1, C)
        mergeSort(S_2, C)
        merge(S_1, S_2, C, S)
```

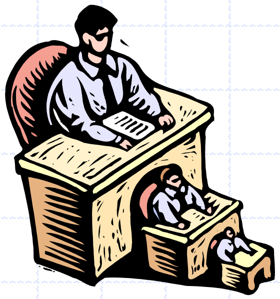
Recurrence Equation Analysis

- The conquer step of merge-sort consists of merging two sorted sequences, each with $n/2$ elements and implemented by means of a doubly linked list, takes at most bn steps, for some constant b.
- Likewise, the basis case ($n < 2$) will take at b most steps.
- Therefore, if we let $T(n)$ denote the running time of merge-sort:

$$T(n) = \begin{cases} b & \text{if } n < 2 \\ 2T(n/2) + bn & \text{if } n \geq 2 \end{cases}$$

- We can therefore analyze the running time of merge-sort by finding a **closed form solution** to the above equation.
  - That is, a solution that has $T(n)$ only on the left-hand side.

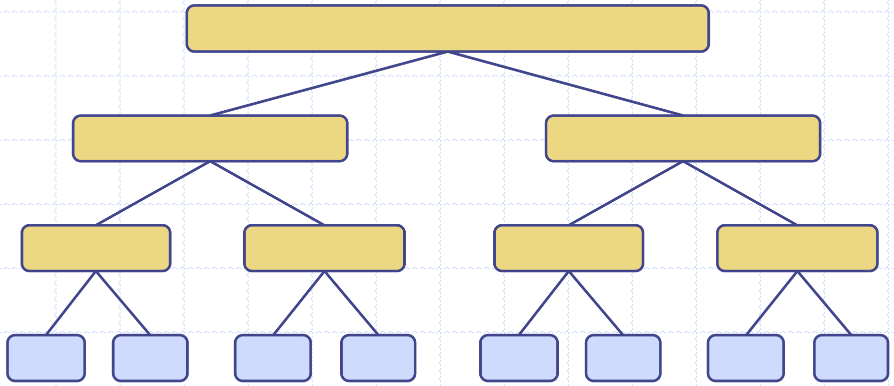
1) The Recursion Tree



- Draw the recursion tree for the recurrence relation and look for a pattern:

$$T(n) = \begin{cases} b & \text{if } n < 2 \\ 2T(n/2) + bn & \text{if } n \geq 2 \end{cases}$$

depth	T's	size	time
0	1	n	bn
1	2	$n/2$	bn
i	2^i	$n/2^i$	bn
…	…	…	…



Total time = $bn + bn \log n$

(last level plus all previous levels)

2) Iterative Substitution

- In the iterative substitution, or “plug-and-chug,” technique, we iteratively apply the recurrence equation to itself and see if we can find a pattern:

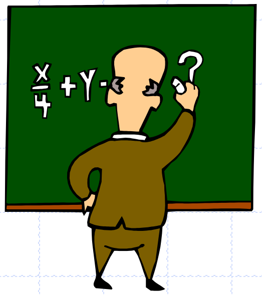
$$\begin{aligned}
T(n) &= 2T(n/2) + bn \\
&= 2(2T(n/2^2)) + b(n/2)) + bn \\
&= 2^2T(n/2^2) + 2bn \\
&= 2^3T(n/2^3) + 3bn \\
&= 2^4T(n/2^4) + 4bn \\
&= \ldots \\
&= 2^iT(n/2^i) + ibn
\end{aligned}$$

- Note that base, T(n)=b, case occurs when $2^i=n$. That is, i = log n.
- So,

$$T(n) = bn + bn\log n$$

- Thus, T(n) is O(n log n).

3) Guess-and-Test Method



- In the guess-and-test method, we guess a closed form solution and then try to prove it is true by induction:

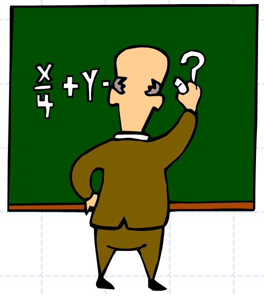
$$T(n) = \begin{cases} b & \text{if } n < 2 \\ 2T(n/2) + bn & \text{if } n \geq 2 \end{cases}$$

- Guess: T(n) $\leq$ cn log n.

- Prove that T(n) $\leq$ c n log n

  for an appropriate choice of c > 0

Guess-and-Test Method



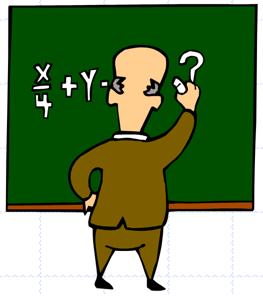
- In the guess-and-test method, we guess a closed form solution and then try to prove it is true by induction:

$$T(n) = \begin{cases} b & \text{if } n < 2 \\ 2T(n/2) + bn & \text{if } n \geq 2 \end{cases}$$

- Guess: $T(n) \leq cn \log n$.

$$\begin{aligned} T(n) &= 2\,T(n/2) + bn \\ &\leq 2(c\,n/2 \log (n/2)) + bn \\ &= cn \log (n/2) + bn \\ &= cn (\log n - \log 2) + bn \\ &= cn (\log n - 1) + bn \\ &= cn \log n - cn + bn \\ &\leq c\,n \log n \quad \text{for } c \geq b \end{aligned}$$

Another Example Guess-and-Test Method

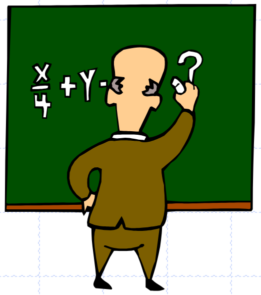


- In the guess-and-test method, we guess a closed form solution and then try to prove it is true by induction:

$$T(n) = \begin{cases} b & \text{if } n < 2 \\ 2T(n/2) + bn\log n & \text{if } n \geq 2 \end{cases}$$

- Guess: T(n) $\leq$ cn log n.

Guess-and-Test Method



- In the guess-and-test method, we guess a closed form solution and then try to prove it is true by induction:

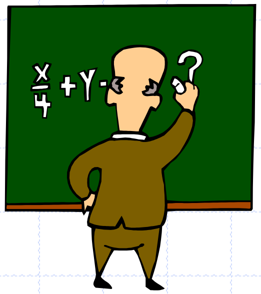
$$T(n) = \begin{cases} b & \text{if } n < 2 \\ 2T(n/2) + bn\log n & \text{if } n \geq 2 \end{cases}$$

- Guess: T(n) $\underline{<}$ cn log n.

$$\begin{aligned} T(n) &= 2T(n/2) + bn \log n \\ &\underline{<} 2(c(n/2)\log(n/2)) + bn \log n \\ &= cn(\log n - \log 2) + bn \log n \\ &= cn \log n - cn + bn \log n \end{aligned}$$

- Wrong: since we cannot make this last line be less than cn log n

Guess-and-Test Method, Guess Again



- Recall the recurrence equation:

$$T(n) = \begin{cases} b & \text{if } n < 2 \\ 2T(n/2) + bn\log n & \text{if } n \geq 2 \end{cases}$$

- Guess #2: T(n) ≤ cn log² n.

$$\begin{aligned} T(n) &= 2T(n/2) + b\,n\log n \\ &\leq 2(c\,(n/2)\log^2(n/2)) + b\,n\log n \\ &= c\,n\,(\log n - \log 2)^2 + b\,n\log n \\ &= c\,n\,(\log^2 n - 2\log n + 1) + b\,n\log n \\ &= c\,n\log^2 n - 2\,c\,n\log n + c\,n + b\,n\log n \\ &\leq c\,n\log^2 n \end{aligned}$$

  - if c > b.
- So, T(n) is O(n log² n).
- In general, to use this method, you need to have a good guess and you need to be good at induction proofs.

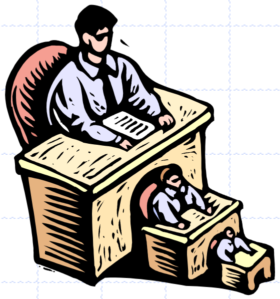
Making a good guess

- No general way
- Takes experience and sometimes creativity
- Some heuristics:
  - If similar to one seen before, then guess a similar solution or
  - Prove loose upper and lower bounds, then raise the lower bound until it converges on the correct solution
    - In our example, prove that O(n) is a lower bound and $O(n^2)$ is a upper bound
- Sometimes the inductive assumption is not strong enough to make the proof work out

Recursion Tree Method

- Draw a recursion tree to help generate a good guess

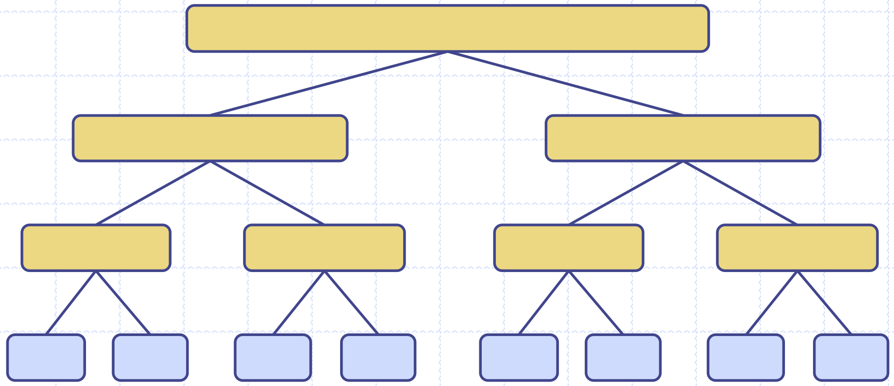
The Recursion Tree



- Draw the recursion tree for the recurrence relation and look for a pattern:

$$T(n) = \begin{cases} b & \text{if } n < 2 \\ 2T(n/2) + bn & \text{if } n \ge 2 \end{cases}$$

depth	T's	size
0	1	n
1	2	$n/2$
i	2^i	$n/2^i$
…	…	…



time
bn
bn
bn
…

Total time = $bn + bn \log n$

(last level plus all previous levels)

Main Point

2. The essential structure of the recurrence relation is brought out by drawing the recurrence tree. The work involved in input splitting and sub-solution combining is indicated on each level and the work of solving the base cases is on the leaves. Such trees provide an intuitive verification of the solution to the recurrence equation.

   Maharishi's Science and Technology of Consciousness provides techniques for verifying the essential structure and nature of pure consciousness through direct experience of one's own Self.

Big-Oh and its Relatives

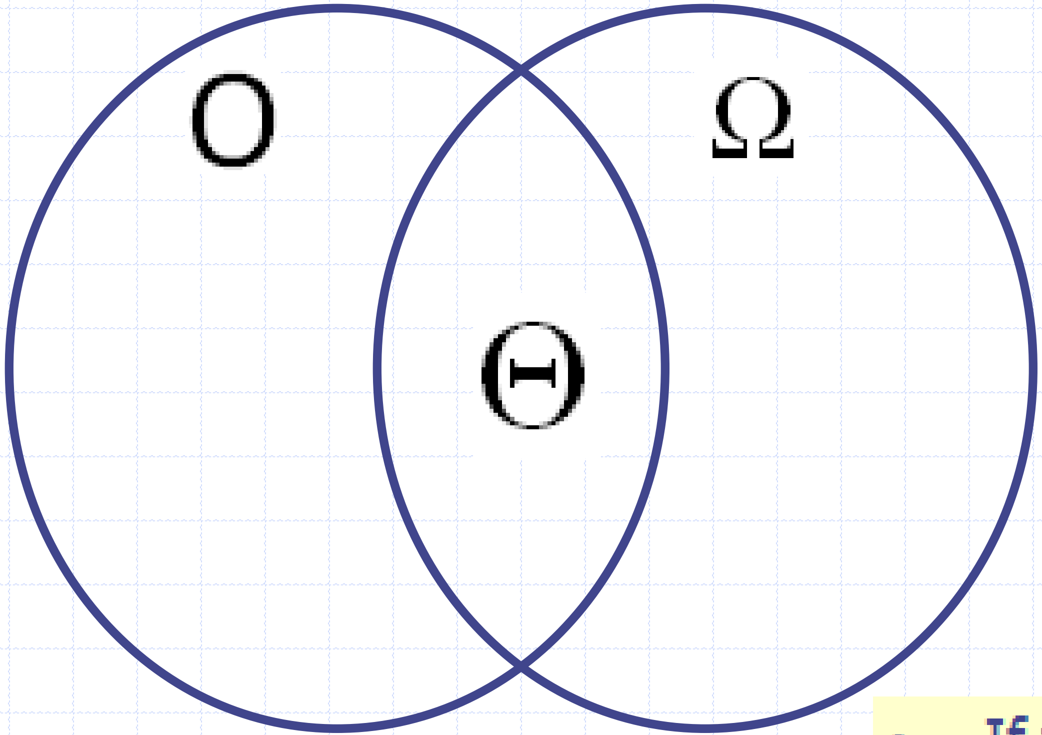
- **big-oh**
  - f(n) is O(g(n)) if, there is a constant $c > 0$ and an integer constant $n_0 > 0$ such that

    $f(n) \leq c \bullet g(n)$ for all $n \geq n_0$
  - f(n) is O(g(n)) if g(n) is an ***asymptotic upper bound*** on f(n)
- **big-Omega**
  - f(n) is $\Omega(g(n))$ if there is a constant $c > 0$ and an integer constant $n_0 \geq 1$ such that

    $f(n) \geq c \bullet g(n)$ for all $n \geq n_0$
  - f(n) is $\Omega(g(n))$ if g(n) is an ***asymptotic lower bound*** on f(n)
- **big-Theta**
  - f(n) is $\Theta(g(n))$ if there are constants $c' > 0$ and $c'' > 0$ and an integer constant $n_0 \geq 1$ such that

    $c' \bullet g(n) \leq f(n) \leq c'' \bullet g(n)$ for all $n \geq n_0$
  - f(n) is $\Theta(g(n))$ if g(n) is an ***asymptotic tight bound*** on f(n)

Relationships Between the Complexity Classes



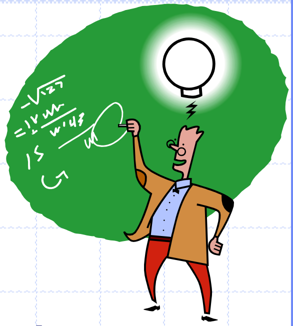
- If f(n) is in both O(g(n)) and Ω(g(n)), it is in Θ(g(n)).

Big-Oh and Growth Rate

- The big-Oh notation gives an upper bound on the growth rate of a function
- The statement "$f(n)$ is $O(g(n))$" means that the growth rate of $f(n)$ is no more than the growth rate of $g(n)$
- We can use the big-Oh notation to rank functions according to their growth rate

	$g(n)$ grows faster	$f(n)$ grows faster
$f(n)$ is $O(g(n))$	Yes	No
$f(n)$ is $\Omega(g(n))$	No	Yes
$f(n)$ is $\Theta(g(n))$	Yes	Yes

Master Method



- Many divide-and-conquer recurrence equations have the form:

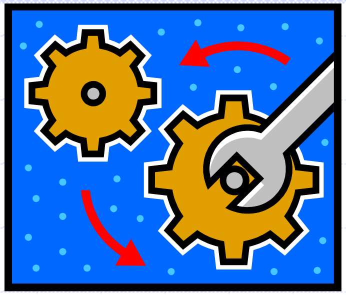
$$T(n) = \begin{cases} c & \text{if } n < d \\ aT(n/b) + f(n) & \text{if } n \geq d \end{cases}$$

- The Master Theorem: for some $\varepsilon > 0$

1. if $f(n)$ is $O(n^{\log_b a - \varepsilon})$, then $T(n)$ is $\Theta(n^{\log_b a})$
2. if $f(n)$ is $\Theta(n^{\log_b a} \log^k n)$, then $T(n)$ is $\Theta(n^{\log_b a} \log^{k+1} n)$
3. if $f(n)$ is $\Omega(n^{\log_b a + \varepsilon})$, then $T(n)$ is $\Theta(f(n))$,

   provided $af(n/b) \leq \delta f(n)$ for some $\delta < 1$.

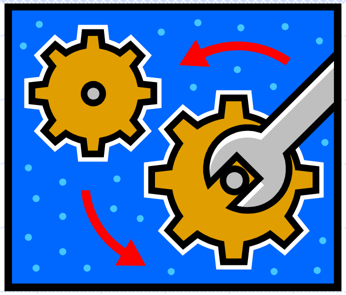
Iterative "Proof" of the Master Theorem



- Using iterative substitution, let us see if we can find a pattern:

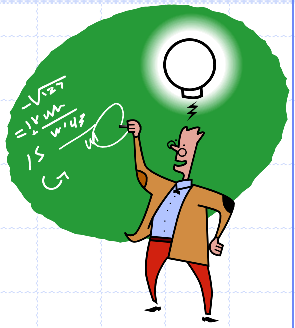
$$
\begin{aligned}
T(n) &= aT(n/b) + f(n) \\
&= a(aT(n/b^2) + f(n/b)) + f(n) \\
&= a^2T(n/b^2) + af(n/b) + f(n) \\
&= a^3T(n/b^3) + a^2 f(n/b^2) + af(n/b) + f(n) \\
&= \ldots \\
&= a^{\log_b n}T(1) + \sum_{i=0}^{(\log_b n)-1} a^i f(n/b^i) \\
&= n^{\log_b a}T(1) + \sum_{i=0}^{(\log_b n)-1} a^i f(n/b^i)
\end{aligned}
$$

Iterative “Proof” of the Master Theorem



- The Master Theorem distinguishes the three cases as
  - The first term is dominant
  - Each part of the summation is equally dominant
  - The summation is a geometric series

Master Method, Example 1



- The form:

$$T(n) = \begin{cases} c & \text{if } n < d \\ aT(n/b) + f(n) & \text{if } n \ge d \end{cases}$$

- The Master Theorem:

$$1.\ \text{if } f(n) \text{ is } O(n^{\log_b a - \varepsilon}), \text{ then } T(n) \text{ is } \Theta(n^{\log_b a})$$

$$2.\ \text{if } f(n) \text{ is } \Theta(n^{\log_b a} \log^k n), \text{ then } T(n) \text{ is } \Theta(n^{\log_b a} \log^{k+1} n)$$

$$3.\ \text{if } f(n) \text{ is } \Omega(n^{\log_b a + \varepsilon}), \text{ then } T(n) \text{ is } \Theta(f(n)),$$

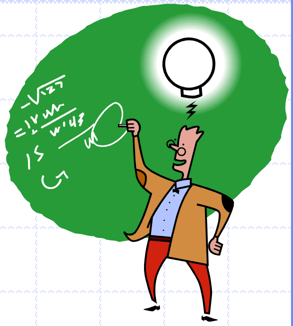
$$\text{provided } af(n/b) \le \delta f(n) \text{ for some } \delta < 1.$$

- Example:

$$T(n) = 4T(n/2) + n$$

Solution: $\log_b a=2$, so case 1 says T(n) is $\Theta(n^2)$.

Master Method, Example 2



- The form:

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

- The Master Theorem:

1. if $f(n)$ is $O(n^{\log_b a - \varepsilon})$, then $T(n)$ is $\Theta(n^{\log_b a})$
2. if $f(n)$ is $\Theta(n^{\log_b a} \log^k n)$, then $T(n)$ is $\Theta(n^{\log_b a} \log^{k+1} n)$
3. if $f(n)$ is $\Omega(n^{\log_b a + \varepsilon})$, then $T(n)$ is $\Theta(f(n))$,

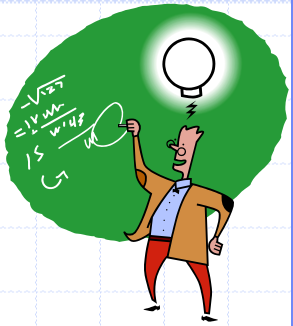
   provided $af(n/b) \leq \delta f(n)$ for some $\delta < 1$.

- Example:

$$T(n) = 2T(n/2) + n \log n$$

Solution: $\log_b a=1$, so case 2 says T(n) is $\Theta(n \log^2 n)$.

Master Method, Example 3



- The form: $$T(n) = \begin{cases} c & \text{if } n < d \\ aT(n/b) + f(n) & \text{if } n \ge d \end{cases}$$

- The Master Theorem:

  1. if $f(n)$ is $O(n^{\log_b a-\varepsilon})$, then $T(n)$ is $\Theta(n^{\log_b a})$
  2. if $f(n)$ is $\Theta(n^{\log_b a}\log^k n)$, then $T(n)$ is $\Theta(n^{\log_b a}\log^{k+1} n)$
  3. if $f(n)$ is $\Omega(n^{\log_b a+\varepsilon})$, then $T(n)$ is $\Theta(f(n))$,

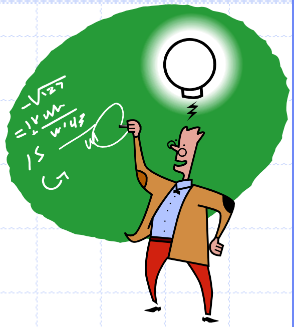
     provided $af(n/b) \le \delta f(n)$ for some $\delta < 1$.

- Example:

$$T(n) = T(n/3) + n\log n$$

Solution: $\log_b a=0$, so case 3 says T(n) is Θ(n log n).

Master Method, Example 4



- The form: $$T(n) = \begin{cases} c & \text{if } n < d \\ aT(n/b) + f(n) & \text{if } n \geq d \end{cases}$$
- The Master Theorem:

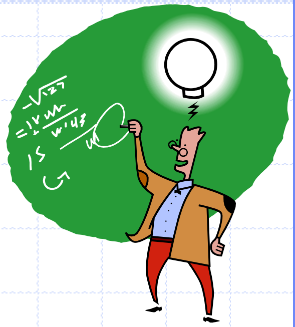
  1. if $f(n)$ is $O(n^{\log_b a - \varepsilon})$, then $T(n)$ is $\Theta(n^{\log_b a})$
  2. if $f(n)$ is $\Theta(n^{\log_b a} \log^k n)$, then $T(n)$ is $\Theta(n^{\log_b a} \log^{k+1} n)$
  3. if $f(n)$ is $\Omega(n^{\log_b a + \varepsilon})$, then $T(n)$ is $\Theta(f(n))$,
     provided $af(n/b) \leq \delta f(n)$ for some $\delta < 1$.

- Example:

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

Solution: $\log_b a=3$, so case 1 says T(n) is $\Theta(n^3)$.

Master Method, Example 5



- The form:

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

- The Master Theorem:

1. if $f(n)$ is $O(n^{\log_b a - \varepsilon})$, then $T(n)$ is $\Theta(n^{\log_b a})$
2. if $f(n)$ is $\Theta(n^{\log_b a} \log^k n)$, then $T(n)$ is $\Theta(n^{\log_b a} \log^{k+1} n)$
3. if $f(n)$ is $\Omega(n^{\log_b a + \varepsilon})$, then $T(n)$ is $\Theta(f(n))$, provided $af(n/b) \leq \delta f(n)$ for some $\delta < 1$.

- Example:

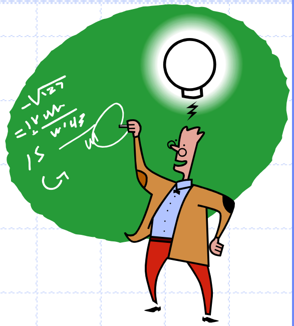
$$T(n) = 9T(n/3) + n^3$$

Solution: $\log_b a=2$, so case 3 says T(n) is $\Theta(n^3)$.

DownHeap Recurrence Equation: $T(n) = T(n/2) + c$

```
Algorithm downHeap(H, size, r)
    Input Array H representing a heap, rank r of an element in H,
          and the size of the heap H
    Output H with the heap property restored

    smallest ← rankOfMin(H, size, r)       {min of r and its children}
    if smallest ≠ r then
        temp ← H[smallest]
        H[smallest] ← H[r]                  {swap elements}
        H[r] ← temp
        downHeap(H, size, smallest)  {one of r's children, half of tree}
```

Master Method, Example 6



- The form:

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

- The Master Theorem:

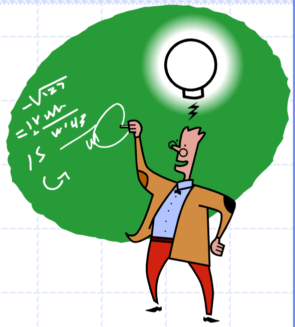
1. if $f(n)$ is $O(n^{\log_b a - \varepsilon})$, then $T(n)$ is $\Theta(n^{\log_b a})$
2. if $f(n)$ is $\Theta(n^{\log_b a} \log^k n)$, then $T(n)$ is $\Theta(n^{\log_b a} \log^{k+1} n)$
3. if $f(n)$ is $\Omega(n^{\log_b a + \varepsilon})$, then $T(n)$ is $\Theta(f(n))$, provided $af(n/b) \leq \delta f(n)$ for some $\delta < 1$.

- Example:

$$T(n) = T(n/2) + 1$$ (downheap, binary search)

Solution: $\log_b a=0$, so case 2 says T(n) is Θ(log n).

Master Method, Example 7



- The form: $$T(n) = \begin{cases} c & \text{if } n < d \\ aT(n/b) + f(n) & \text{if } n \geq d \end{cases}$$

- The Master Theorem:

  1. if $f(n)$ is $O(n^{\log_b a - \varepsilon})$, then $T(n)$ is $\Theta(n^{\log_b a})$
  2. if $f(n)$ is $\Theta(n^{\log_b a} \log^k n)$, then $T(n)$ is $\Theta(n^{\log_b a} \log^{k+1} n)$
  3. if $f(n)$ is $\Omega(n^{\log_b a + \varepsilon})$, then $T(n)$ is $\Theta(f(n))$,
     provided $af(n/b) \leq \delta f(n)$ for some $\delta < 1$.

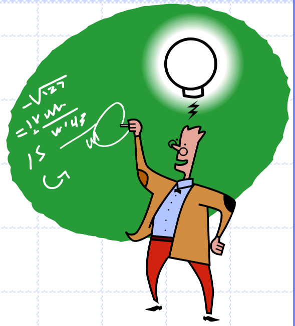
- Example:

$$T(n) = T(3n/4) + 2bn$$

(quick select)

Solution: $\log_b a$=0, so case 3 says T(n) is Θ(n).

Master Method, Example 8



- The form: $$T(n) = \begin{cases} c & \text{if } n < d \\ aT(n/b) + f(n) & \text{if } n \geq d \end{cases}$$
- The Master Theorem:

$$1.\ \text{if } f(n) \text{ is } O(n^{\log_b a - \varepsilon}), \text{ then } T(n) \text{ is } \Theta(n^{\log_b a})$$

$$2.\ \text{if } f(n) \text{ is } \Theta(n^{\log_b a} \log^k n), \text{ then } T(n) \text{ is } \Theta(n^{\log_b a} \log^{k+1} n)$$

$$3.\ \text{if } f(n) \text{ is } \Omega(n^{\log_b a + \varepsilon}), \text{ then } T(n) \text{ is } \Theta(f(n)),$$

$$\text{provided } af(n/b) \leq \delta f(n) \text{ for some } \delta < 1.$$

- Example:

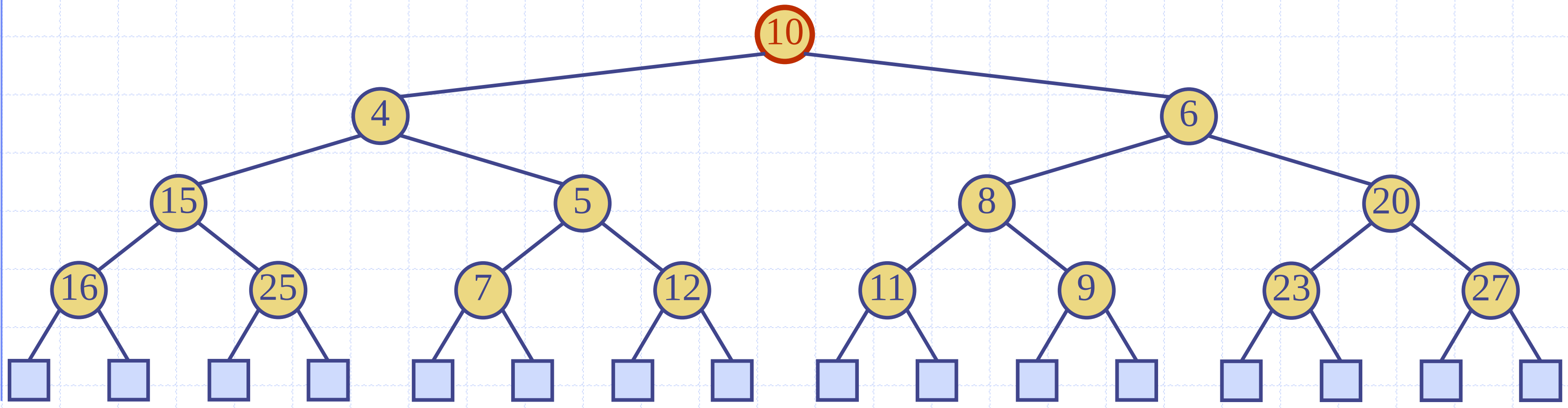
$$T(n) = T(n/2) + \log n$$

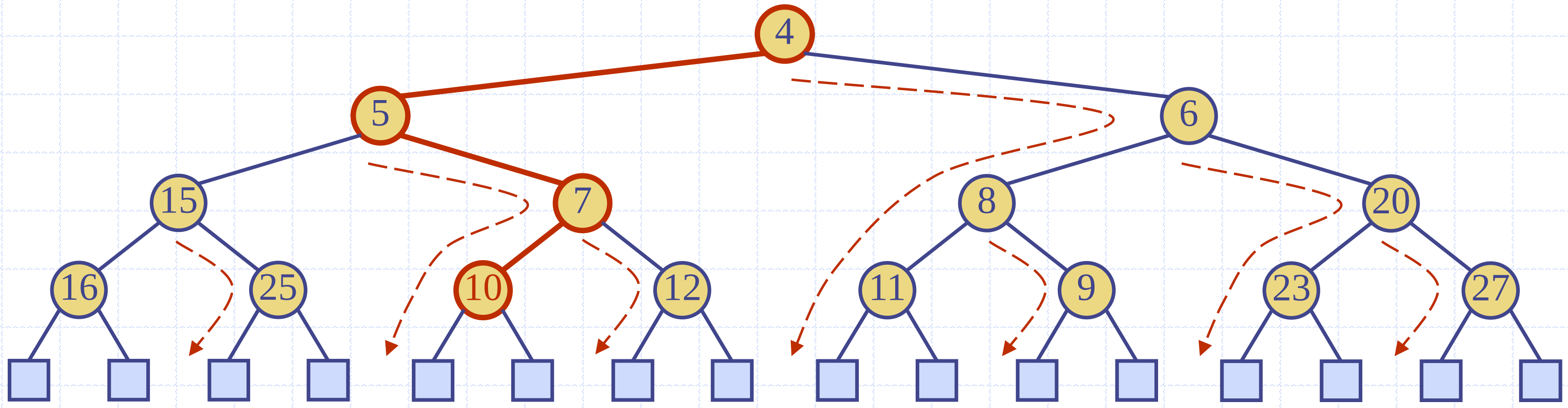
Solution: $\log_b a=0$, so case 2 says T(n) is $\Theta(\log^2 n)$.

Build the Heap from bottom using a while-loop

```
Algorithm buildHeap(arr)
   Input Array arr
   Output arr is a heap built from the bottom up in O(n) time
                              with the root at index 0 (instead of 1)
        last ← arr.length-1;
        next ← last;
        while (next > 0) do
                downHeap(arr, last, parent(next));
                next ← next - 2;

Algorithm parent(i)
        return floor((i  – 1)/ 2)
```

Example (end)





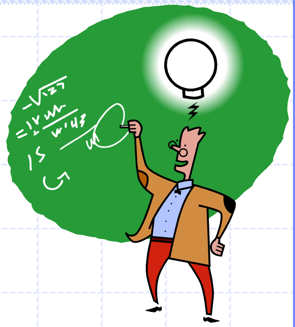
Build the Heap from bottom Translated from Interation

```
Algorithm buildHeap(arr, next)
  Input Array arr
  Output arr is a heap built from the bottom up in O(n) time
                  with the root at index 0 (instead of 1)
     last ← arr.length-1;
     if (next > 0) then
          downHeap(arr, last, parent(next));
          buildHeap(arr, next - 2)

Algorithm parent(i)
     return floor((i  – 1)/ 2)
```

Build the Heap from bottom up using Divide-and-Conquer

```
Algorithm buildHeap(arr, next)
   Input Array arr, next is the rank of current subtree root
   Output Array arr is a heap built from the bottom up in O(n) time
                         with the root at index 0 (instead of 1)
      last ← arr.length-1;
      if  next ≤ parent(last)  then
            left ← (next * 2) + 1          // left child of next when root is at 0
            right ← left + 1               // right child of next
            buildHeap(arr, left)           // half of the balanced tree/heap T(n/2)
            buildHeap(arr, right)          // half of the balanced tree/heap T(n/2)
            downHeap(arr, last, next);     // log n

Algorithm parent(i)
      return floor((i  – 1)/ 2)            // parent when root is at 0
```

Master Method, Example 9



- The form: $$T(n) = \begin{cases} c & \text{if } n < d \\ aT(n/b) + f(n) & \text{if } n \ge d \end{cases}$$
- The Master Theorem:

$$1.\ \text{if } f(n) \text{ is } O(n^{\log_b a - \varepsilon}), \text{ then } T(n) \text{ is } \Theta(n^{\log_b a})$$

$$2.\ \text{if } f(n) \text{ is } \Theta(n^{\log_b a} \log^k n), \text{ then } T(n) \text{ is } \Theta(n^{\log_b a} \log^{k+1} n)$$

$$3.\ \text{if } f(n) \text{ is } \Omega(n^{\log_b a + \varepsilon}), \text{ then } T(n) \text{ is } \Theta(f(n)),$$

$$\text{provided } af(n/b) \le \delta f(n) \text{ for some } \delta < 1.$$

- Example:

$$T(n) = 2T(n/2) + \log n$$ (heap construction)

Solution: $\log_b a=1$, so case 1 says T(n) is Θ(n).

Main Point

1. Divide-and-conquer algorithms can be directly translated into a recurrence relation; this can then be translated directly into a precise estimate of the algorithm's time complexity. Mathematics forms the basis of these analytic techniques (e.g., the Master Theorem). Their proofs of validity give us confidence in their correctness.
   Maharishi's Science and Technology of Consciousness provides systematic techniques for experiencing total knowledge of the Universe to enhance individual life.

Connecting the Parts of Knowledge with the Wholeness of Knowledge

1. Divide-and-conquer algorithms can be directly translated from the code into a recurrence relation that can then be solved by iterative substitution or a Guess-and-Test inductive proof.

2. Through mathematical proof (using iterative substitution), it has been shown that a recurrence relation can be solved much more easily using the Master Theorem. The three cases use asymptotic bound techniques that compare function growth, big-O, big-Omega, and big-Theta.

3. <u>**Transcendental Consciousness**</u>, when directly experienced, is the basis for fully understanding the unified field located by Physics.
4. <u>**Impulses within Transcendental Consciousness**</u>: The dynamic natural laws within this field create and maintain the order and balance in creation. We verify this through regular practice and finding the nourishing influence of the Absolute in all areas of our life.
5. <u>**Wholeness moving within itself**</u>**:** In Unity Consciousness, knowledge is on the move; the fullness of pure consciousness is flowing onto the outer fullness of relative experience. Here there is nothing but knowledge; the knowledge is self-validating.