

# Master Method

Goal: Find the solution to recurrences of the form  $T(n) = a T(n/b) + f(n)$

closed form solution

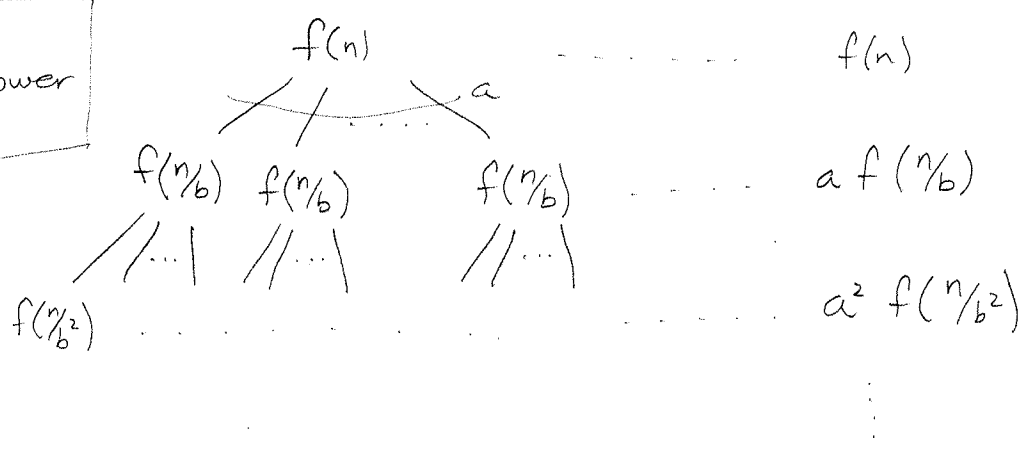
$T(1) = \Theta(1)$  this can have  $\lfloor \rfloor$  or  $\lceil \rceil$

Why? These recurrences often arise in algorithm analysis:

- divide a problem into "a" subproblems each of size  $n/b$
- use  $f(n)$  time to do the dividing of the problem and combining the results of subproblems

Try recursion tree:

Assume  $n$  is a power of  $b$



leaves  $\Theta(1)$   $\Theta(1)$  ...  $\Theta(1)$  ...  $\Theta(a^{\log_b n})$

$$T(n) = \sum_{i=0}^{\log_b n - 1} a^i f(n/b^i) + \underbrace{\Theta(n^{\log_b a})}_{\text{leaves}}$$

Now what?

# Master Theorem

(17)

For  $T(n) = a T(n/b) + f(n)$ ,  $T(1) = \Theta(1)$

Case 1  $f(n) = O(n^{\log_b a - \epsilon})$  for some constant  $\epsilon > 0$

Then  $T(n) = \Theta(n^{\log_b a})$  [leaf cost dominates]

Case 2  $f(n) = \Theta(n^{\log_b a})$

Then  $T(n) = \Theta(n^{\log_b a} \lg n)$ . [every level contributes cost  $n^{\log_b a}$ ]

Case 3  $f(n) = \Omega(n^{\log_b a + \epsilon})$  for some  $\epsilon > 0$  and  
 $a f(n/b) \leq c f(n)$  for  $c < 1$  for all  $n > n_0$ .

Then  $T(n) = \Theta(f(n))$  [root cost dominates]

Cases compare  $f(n)$  to  $n^{\log_b a}$

If  $f(n) \boxed{<} n^{\log_b a - \epsilon}$  then  $T(n) = \Theta(n^{\log_b a})$

If  $f(n) \boxed{=} n^{\log_b a}$  then  $T(n) = \Theta(n^{\log_b a} \lg n)$

If  $f(n) \boxed{>} n^{\log_b a + \epsilon}$  then\*  $T(n) = \Theta(f(n))$

by a polynomial amount

asymptotically

\* Note: we also need  $a f(n/b) \leq c f(n)$  for some  $c < 1$

Note also:  $a f(n/b) \leq c f(n)$  for  $c < 1$  implies  $f(n) \boxed{>} n^{\log_b a + \epsilon}$

For example:  $T(n) = 9T(n/3) + n$

$a=9$   $b=3$   $f(n)=n$   $n^{\log_b a} = n^{\log_3 9} = n^2$

$f(n)=n \in O(n^{2-\epsilon})$  for  $\epsilon=1$  so... Case 1

$T(n) = \Theta(n^2)$

Ex:  $T(n) = 3T(n/4) + n \lg n$

$a=3 \quad b=4 \quad f(n) = n \lg n \quad n^{\log_b a} = n^{\log_4 3} \leq n^{0.793}$

Since  $f(n) \in \Omega(n^{\log_4 3 + \epsilon})$  for  $\epsilon = 0.2$

and  $a f(n/b) = 3 \cdot (n/4) \lg(n/4) \leq (3/4) n \lg n = c f(n)$

for  $c = 3/4$ , Case 3

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

The three cases don't cover all possibilities:

$T(n) = 2T(n/2) + n \lg n$  has proper form but

$a=2 \quad b=2 \quad f(n) = n \lg n \quad n^{\log_b a} = n$

so  $f(n) \in \Omega(n)$  but  $f(n) \notin \Omega(n^\epsilon)$  for  $\epsilon > 0$

so case 3 doesn't apply.

### Readings

Ch 7 Quicksort

8 Sort lower bound

9 Selection

# Back to the Skyline problem . . . .

We have an algorithm that runs in  $\Theta(n \log n)$  time.  
Is there a faster algorithm for this problem?

Let  $a$  be the fastest algorithm for Skyline problem.

Idea: Use  $a$  to build a sorting algorithm

$\text{SkySort}(A[1..n])$       left   height   right  
 for  $i = 1$  to  $n$       ↙      ↘      ↘  
 $B[i] = (A[i], A[i], A[i]+1)$

$S = a(B)$  ← get skyline  $S = ((x_1, y_1)(x_2, y_2) \dots (x_m, y_m))$

for  $i = 1$  to  $m$

If  $x_i = y_i$  then print  $x_i$

SkySort is a comparison based sorting algorithm.

so it must take time  $\geq c \cdot n \log n$       WHY?

SkySort takes time  $\leq d \cdot n + T_a(n)$

Thus

$$T_a(n) + d \cdot n \geq c \cdot n \log n$$

$$\Rightarrow T_a(n) \geq c \cdot n \log n - d \cdot n$$

$$\in \Omega(n \log n)$$

only can access input by comparing  $i^{\text{th}}$  element to  $j^{\text{th}}$  element.

little  $o$

fast  $\downarrow$   
 $\equiv o(n \log n)$   
in this case

Using an algorithm for Skyline to solve Sort is called reducing Sort to Skyline.

We've written a reduction (an algorithm) to solve Sort given any algorithm for Skyline. The reduction is fast so a fast algorithm for Skyline  $\Rightarrow$  a fast sorting algorithm

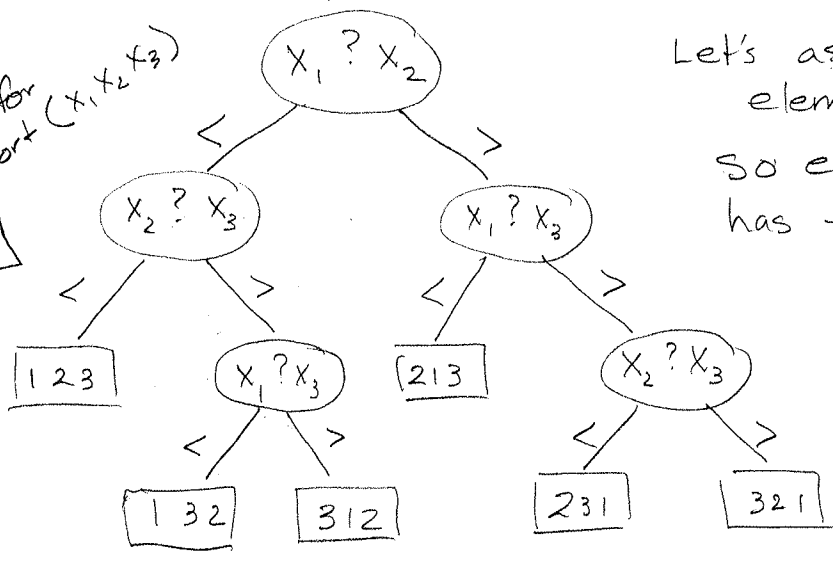
Claim Any comparison based algorithm to sort  $n$  elements makes  $\geq cn \log n$  comparisons. (for positive constant  $c$ )

Proof Any algorithm can be model as a decision tree.

- The nodes of the tree correspond to comparisons made by the algorithm (the root is the first comparison)

between input elements

Decision tree for Insertion Sort  $(x_1, x_2, x_3)$



Let's assume all input elements are distinct. So each comparison has two outcomes " $<$ " and " $>$ "

- The algorithm's next comparison depends on the outcome of the previous comparisons

Eventually the algorithm must output the correct sorted order.

If  $M = \#$  possible outputs (i.e. orderings  $M = n!$ ) then  $\#$  leaves of decision tree  $\geq M$

Worst case  $\#$  comparisons = depth of tree = <sup>length of</sup> longest root-leaf path

$$M \leq \# \text{leaves} \leq 2^{\text{depth}} \Rightarrow \text{depth} \geq \log_2 M$$

$$\Rightarrow \text{depth} \geq \log_2(n!) \in \Omega(n \log n).$$

What is a non-comparison based sort?

(21)

## Bucket / Counting Sort

If we know the keys (of the elements we are sorting) are in the set  $\{0, 1, \dots, k-1\}$  we can use them to index into an array  $B[0 \dots k-1]$  of buckets (queues)

### Example

0	
1	①
2	△ 2
3	③ △
⋮	
k	

### Bucket Sort ( $A[1 \dots n]$ )

$B$  = array of  $k$  empty buckets (queues)

for  $i = 1$  to  $n$

[ Enqueue  $A[i]$  into bucket  $B[A[i].key]$

$i = 1$

for  $j = 0$  to  $k-1$

[ While  $B[j]$  not empty

[  $S[i++] =$  Dequeue  $B[j]$

Bucket Sort (△ ③ 2 ① △)

⇒ ① △ 2 ③ △ return  $S$

Running time  $O(n+k)$  if  $k$  is small (meaning  $O(n)$ ) then running time is  $O(n)$ .

Note: Bucket sort is a stable sort meaning items with the same key appear in the output in the same order as they appear in the input.

## Radix Sort

Assume keys are  $d$  digit numbers where each digit takes on  $k$  possible values.

### Radix Sort ( $A[1 \dots n]$ )

[digit 1 is least significant]

for  $i = 1$  to  $d$

$A =$  Bucket Sort ( $A[1 \dots n]$ ) using  $i^{\text{th}}$  digit of key as key for Bucket Sort

### Example

Radix Sort (321, 122, 121)

$\begin{array}{|l} 321 \\ 122 \\ 121 \end{array} \Rightarrow \begin{array}{|l} 321 \\ 121 \\ 122 \end{array} \Rightarrow \begin{array}{|l} 321 \\ 121 \\ 122 \end{array} \Rightarrow \begin{array}{|l} 121 \\ 122 \\ 321 \end{array}$

Running time

$O(d(n+k))$