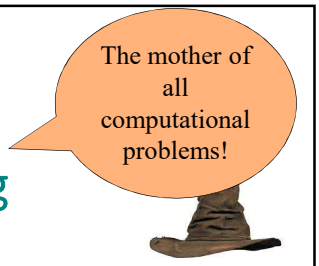
Sorting



6.006 Lecture 3

Image from pottermore.com

Sorting

ordering

alphabetizing

Why?

Easier to find things later!

Images from www.davie-fl.gov and enel460crawford.blogspot.com

Some uses of sorting:

- Problems that become easy once items are in sorted order
  - Searching: Is "Erik Demaine" in the database?
  - Statistics: Identify statistical outliers, or a median
- Non-obvious applications
  - Find closest pair
  - Data compression: sorting finds duplicates
  - Computer graphics: rendering scenes front to back

Today's plan

- Sorting definition
- Two sorting algorithms
  - Insertion Sort
  - Merge Sort
- Solving Recurrences

The problem of sorting

Input: array A[1…n] of numbers.

Output: permutation B[1…n] of A such that $B[1] \le B[2] \le \cdots \le B[n]$

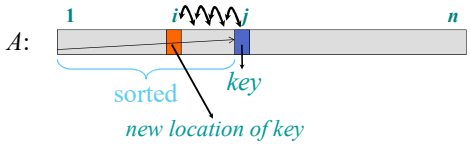
e.g. A = [7, 2, 5, 5, 9.6] → B = [2, 5, 5, 7, 9.6]

How can we do it efficiently ?

Insertion sort

INSERTION-SORT (A, n) ▷ $A[1 \,..\, n]$
for $j \leftarrow 2$ **to** n
  insert key $A[j]$ **into the (already sorted) sub-array** $A[1 \,..\, j\text{-}1]$**.**
  by pairwise key-swaps down to its sorted position

Illustration of iteration j



Example of insertion sort

8 2 4 9 3 6

Example of insertion sort

8 2 4 9 3 6

Example of insertion sort

8 2 4 9 3 6

2 8 4 9 3 6

Example of insertion sort

8 2 4 9 3 6

2 8 4 9 3 6

Example of insertion sort

8 2 4 9 3 6

2 8 4 9 3 6

2 4 8 9 3 6

Example of insertion sort

8 2 4 9 3 6

2 8 4 9 3 6

2 4 8 9 3 6

Example of insertion sort

8 2 4 9 3 6
2 8 4 9 3 6
2 4 8 9 3 6
2 4 8 9 3 6

Example of insertion sort

8 2 4 9 3 6
2 8 4 9 3 6
2 4 8 9 3 6
2 4 8 9 3 6

Example of insertion sort

8 2 4 9 3 6
2 8 4 9 3 6
2 4 8 9 3 6
2 4 8 9 3 6
2 3 4 8 9 6

Example of insertion sort

8 2 4 9 3 6
2 8 4 9 3 6
2 4 8 9 3 6
2 4 8 9 3 6
2 3 4 8 9 6

Example of insertion sort

8 2 4 9 3 6
2 8 4 9 3 6
2 4 8 9 3 6
2 4 8 9 3 6
2 3 4 8 9 6
2 3 4 6 8 9 *done*

Running time:

- Upper bound $O(n^2)$
  - n times through loop
  - at most n swaps in each loop
- Algorithm will need $\Omega(n^2)$
  - E.g. Input in reverse sorted order n, n-1, n-2,…
    - i comparisons and swaps in loop i
    - Total $\Sigma i = \frac{n(n-1)}{2} = \theta(n^2)$

A nice property of insertion sort:

"in place sorting": A gets modified into B
without using additional memory!!

An improvement? Binary Insertion sort

BINARY-INSERTION-SORT (A, n) ▷ $A[1 \,..\, n]$
for $j \leftarrow 2$ **to** n
insert key $A[j]$ **into the (already sorted) sub-array** $A[1 \,..\, j\text{-}1]$.
Use binary search to find the right position

Binary search takes only $\Theta(\log n)$ time.
But, shifting elements after insertion still takes $\Theta(n)$ time!

Complexity: $\Theta(n \log n)$ comparisons
$\Theta(n^2)$ total time

Another approach: Divide and conquer

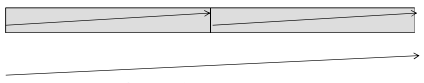
- Divide input into parts
- Conquer (solve) each part recursively
- Combine results to solve original

Run time:
$T(n) = divide\ time$
$+T(n_1) + T(n_2) + \ \ldots + T(n_k)$
$+\ combine\ time$

Meet Merge Sort

divide and conquer

MERGE-SORT $A[1 \,..\, n]$
1. If $n = 1$, done (nothing to sort).
2. Otherwise, recursively sort $A[\ 1 \,..\, n/2\ ]$ and $A[\ n/2{+}1 \,..\, n\ ]$.
3. "***Merge***" the two sorted sub-arrays.



Key subroutine: **MERGE**

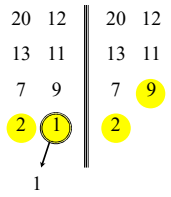
Merging two sorted arrays

20 12
13 11
7 9
2 1

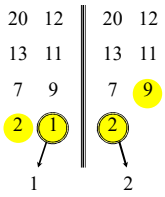
Merging two sorted arrays

20 12
13 11
7 9
2 1
1

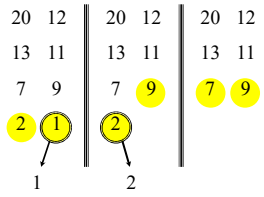
Merging two sorted arrays



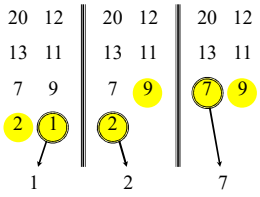
Merging two sorted arrays



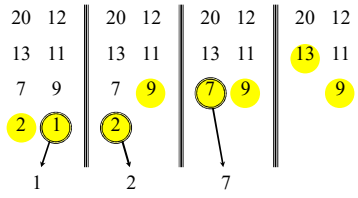
Merging two sorted arrays



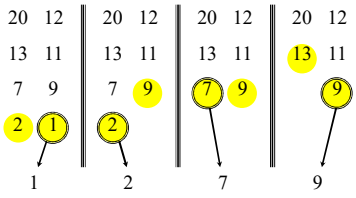
Merging two sorted arrays



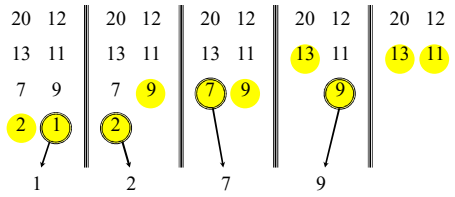
Merging two sorted arrays



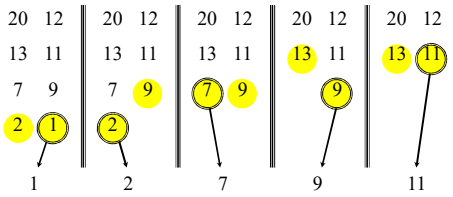
Merging two sorted arrays



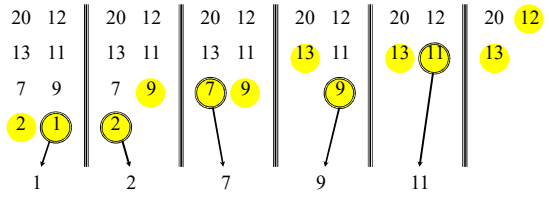
Merging two sorted arrays



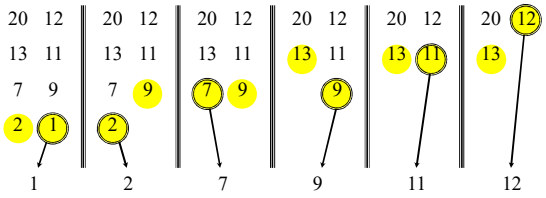
Merging two sorted arrays



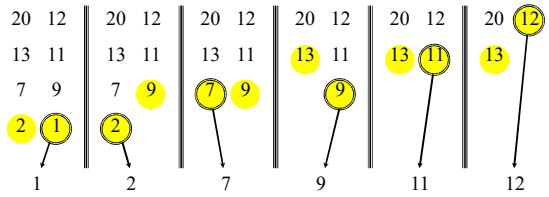
Merging two sorted arrays



Merging two sorted arrays



Merging two sorted arrays



Time = $\Theta(n)$ to merge a total of n elements (linear time).

Analyzing merge sort

MERGE-SORT $A[1 \,..\, n]$

1. If $n = 1$, done. — $T(n)$ = $\Theta(1)$
2. Recursively sort $A[\,1 \,..\, \lceil n/2 \rceil\,]$ and $A[\,\lceil n/2 \rceil + 1 \,..\, n\,]$. — $2T(n/2)$
3. *"Merge"* the two sorted lists — $\Theta(n)$

$$T(n) = \begin{cases} \Theta(1) \text{ if } n = 1; \\ 2T(n/2) + \Theta(n) \text{ if } n > 1. \end{cases}$$

$T(n) = ?$

Recurrence solving

Solve $T(n) = 2T(n/2) + cn$, where $c > 0$ is constant.

Recursion tree method

- Running time represented by sum of nodes
  - Describes where the work happens in the recursive call structure
- Will continually modify the tree in ways that
  - PRESERVE the sum
  - Make it progressively "simpler"
  - Note that "recursive calls" are only at leaves

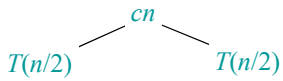
Recursion tree

Solve $T(n) = 2T(n/2) + cn$, where $c > 0$ is constant.

$T(n)$

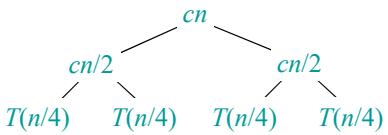
Recursion tree

Solve $T(n) = 2T(n/2) + cn$, where $c > 0$ is constant.



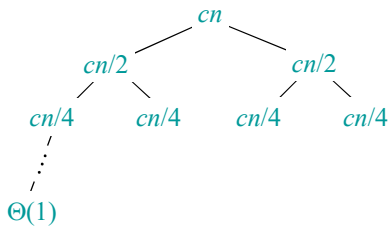
Recursion tree

Solve $T(n) = 2T(n/2) + cn$, where $c > 0$ is constant.



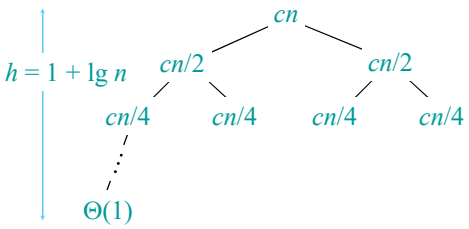
Recursion tree

Solve $T(n) = 2T(n/2) + cn$, where $c > 0$ is constant.



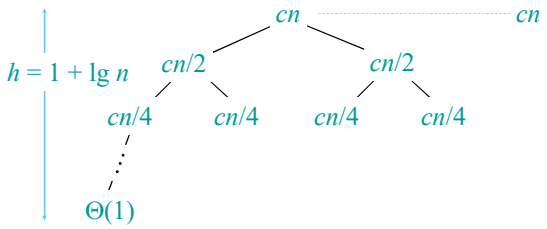
Recursion tree

Solve $T(n) = 2T(n/2) + cn$, where $c > 0$ is constant.



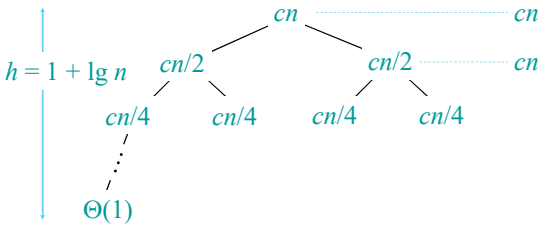
Recursion tree

Solve $T(n) = 2T(n/2) + cn$, where $c > 0$ is constant.



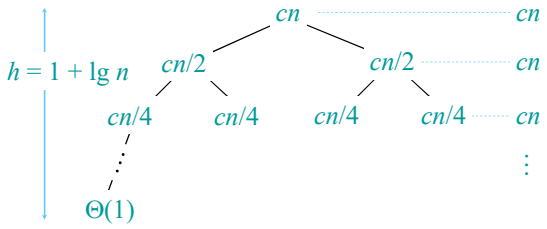
Recursion tree

Solve $T(n) = 2T(n/2) + cn$, where $c > 0$ is constant.



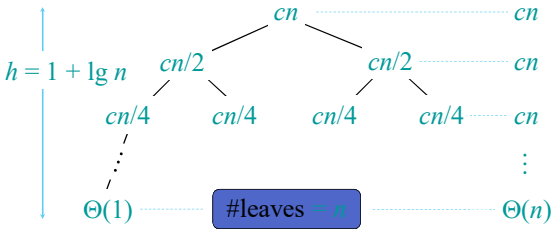
Recursion tree

Solve $T(n) = 2T(n/2) + cn$, where $c > 0$ is constant.



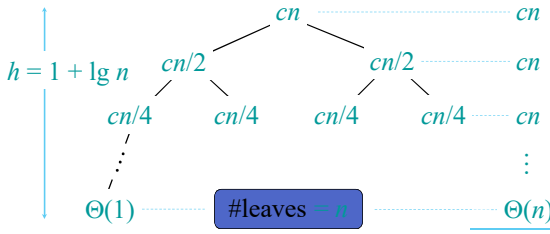
Recursion tree

Solve $T(n) = 2T(n/2) + cn$, where $c > 0$ is constant.



Recursion tree

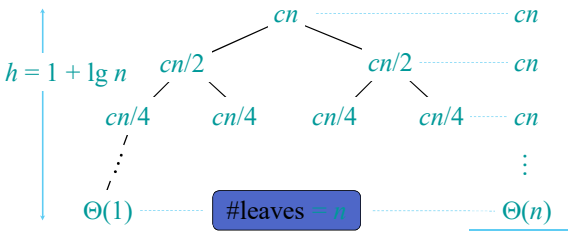
Solve $T(n) = 2T(n/2) + cn$, where $c > 0$ is constant.



Total ?

Recursion tree

Solve $T(n) = 2T(n/2) + cn$, where $c > 0$ is constant.

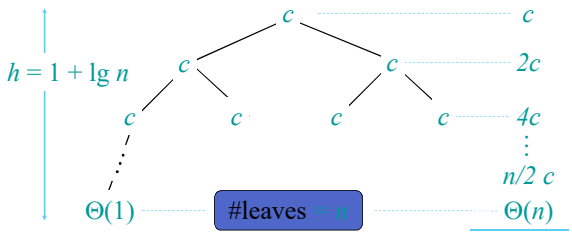


Total = $\Theta(n \lg n)$

Equal amount of work done at each level

Tree for different recurrence

Solve $T(n) = 2T(n/2) + c$, where $c > 0$ is constant.



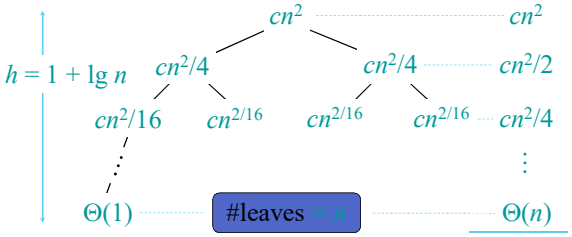
Note that $1 + ½ + ¼ + \ldots < 2$

Total = $\Theta(n)$

All the work done at the leaves

Tree for yet another recurrence

Solve $T(n) = 2T(n/2) + cn^2$, $c > 0$ is constant.



Note that $1 + ½ + ¼ + \ldots < 2$

Total = $\Theta(n^2)$

All the work done at the root

Fun fact:

- If level sums "same", total = $\theta(level\ sum \cdot number\ of\ levels)$
- If level sums geometrically decreasing, total = $\theta(top\ level) = \theta(root\ node)$
- If level sums geometrically increasing, total = $\theta(bottom\ level) = \theta(number\ of\ leaves)$

Master theorem

- A recipe for solving a typical form of divide and conquer recurrence
- Formalizes the "fun fact"
- A bit messy to get used to, use with care!

Master Theorem: T(n) = a T(n/b) + f(n)

- Some observations about recursion tree:
  - h = number of levels = $\log_b n = O(\log n)$
  - L = number of leaves = $a^h = a\text{^}(\log_b n) = n\text{^}(\log_b a)$
- Case 1: Geometrically increasing (e.g. $T(n) = 2T(n/2) + c$)

  If $f(n) = O(L^{1-\epsilon})$ =O($n\text{^}((\log_b a) - \epsilon)$)
  then $T(n) = \theta(L) = \theta(n\text{^}(\log_b a))$

Master Theorem: T(n) = a T(n/b) + f(n)

- Some observations about recursion tree:
  - h = number of levels = $\log_b n = O(\log n)$
  - L = number of leaves = $a^h = a\text{^}(\log_b n) = n\text{^}(\log_b a)$
- Case 2: Equal levels (e.g. *T(n) = 2T(n/2)+cn)*
  If $f(n) = \theta(L) = O(n\text{^}(\log_b a))$ then $T(n) = \theta(L \log n) = \theta(n\text{^}(\log_b a) \cdot \log n)$

Master Theorem: T(n) = a T(n/b) + f(n)

- Some observations about recursion tree:
  - h = number of levels = $\log_b n = O(\log n)$
  - L = number of leaves = $a^h = a\text{^}(\log_b n) = n\text{^}(\log_b a)$
- Case 3: Geometrically decreasing (e.g. *T(n) = 2T(n/2) + cn^2)*
  If $f(n) = O(L^{1+\epsilon}) = O(n\text{^}((\log_b a) + \epsilon))$ and $af\left(\frac{n}{b}\right) \leq (1-\delta)f(n)$ then $T(n) = \theta(f(n))$

Why does it work?

- See textbook for proof

Distinction to note:

Running time of algorithm $\theta(n^2)$

vs.

Time required by a problem $O(n \log n)$