Math 182 Lecture 6

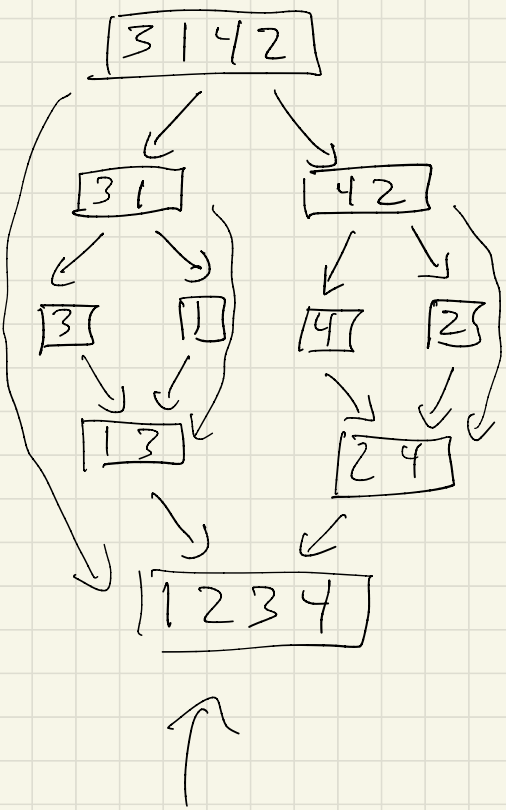
Merge-Sort Running Time

subroutine

```
MERGE-SORT(A, p, r)
1  if p < r
2      q = ⌊(p + r)/2⌋
3      MERGE-SORT(A, p, q)
4      MERGE-SORT(A, q + 1, r)
5      MERGE(A, p, q, r)
```

div. | comq. | comb-

```
MERGE(A, p, q, r)
 1  n_1 = q − p + 1                          } constant
 2  n_2 = r − q                                          ← linear
 3  let L[1 .. n_1 + 1] and R[1 .. n_1 + 1] be new arrays
 4  for i = 1 to n_1                         }
 5      L[i] = A[p + i − 1]                  }
 6      // Copies A[p .. q] into L[1 .. n_1] }  n
 7  for j = 1 to n_1                         }
 8      R[j] = A[q + j]                      }
 9      // Copies A[q + 1 .. r] into R[1 .. n_1] }
10  L[n_1 + 1] = ∞                           } constant
11  R[n_2 + 1] = ∞                           }
12  i = 1                                    }
13  j = 1                                    }
14  for k = p to r                           }
15      if L[i] ≤ R[j]                       } constant  } n times
16          A[k] = L[k]                      }           }
17          i = i + 1                        }           }
18      else A[k] = R[j]                     }           }
19          j = j + 1                        }           }
```



What is running time of Merge?

Sps $n := r - p + 1$ is total size of input.

Merge has running time $\Theta(n)$.

Merge-Sort is an example of a "divide-and-conquer" algorithm.
It involves essentially 3 steps:

Divide: Divide n-element array into two $n/2$-element subarrays

Conquer: Recursively call Merge-Sort to sort each subarray separately

Combine: Used Merge to combine two sorted subarrays into sorted array

---

Sps $n=1$ then running time $T(n)$ is constant:

$$T(n) = \Theta(1) \quad \text{if } n=1$$

Sps $n \geq 2$

Divide: To compute q takes constant time line 2 takes $\Theta(1)$

Conquer: Each rec call to Merge Sort takes $T(n/2)$ so $T(n/2)+T(n/2)$ $= 2T(n/2)$ (really: $T(\lceil n/2 \rceil)+T(\lfloor n/2 \rfloor)$ but we'll assume n is even and even assume n is a power of 2)

Conquer: $\Theta(n)$

Summarize! $n \geq 2$ $T(n) = \Theta(1) + 2T(n/2) + \Theta(n)$

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

We get recurrence

$$T(n) = \begin{cases} \Theta(1) & \text{if } n = 1 \\ 2T(n/2) + \Theta(n) & \text{if } n \geq 2 \end{cases}$$

# of subproblems

size of each subproblem compared to original problem

work done to divide/combine

Can rewrite as:

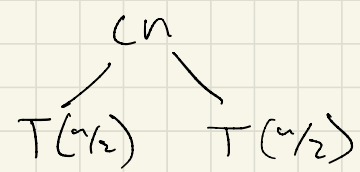
$$T(n) = \begin{cases} c & \text{if } n = 1 \\ 2T(n/2) + cn & \text{if } n \geq 2 \end{cases}$$

will analyze this recurrence.

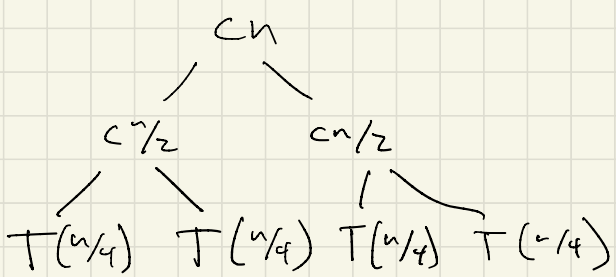
Will use recursion tree method

(Assume n is power of 2.)

$T(n)$

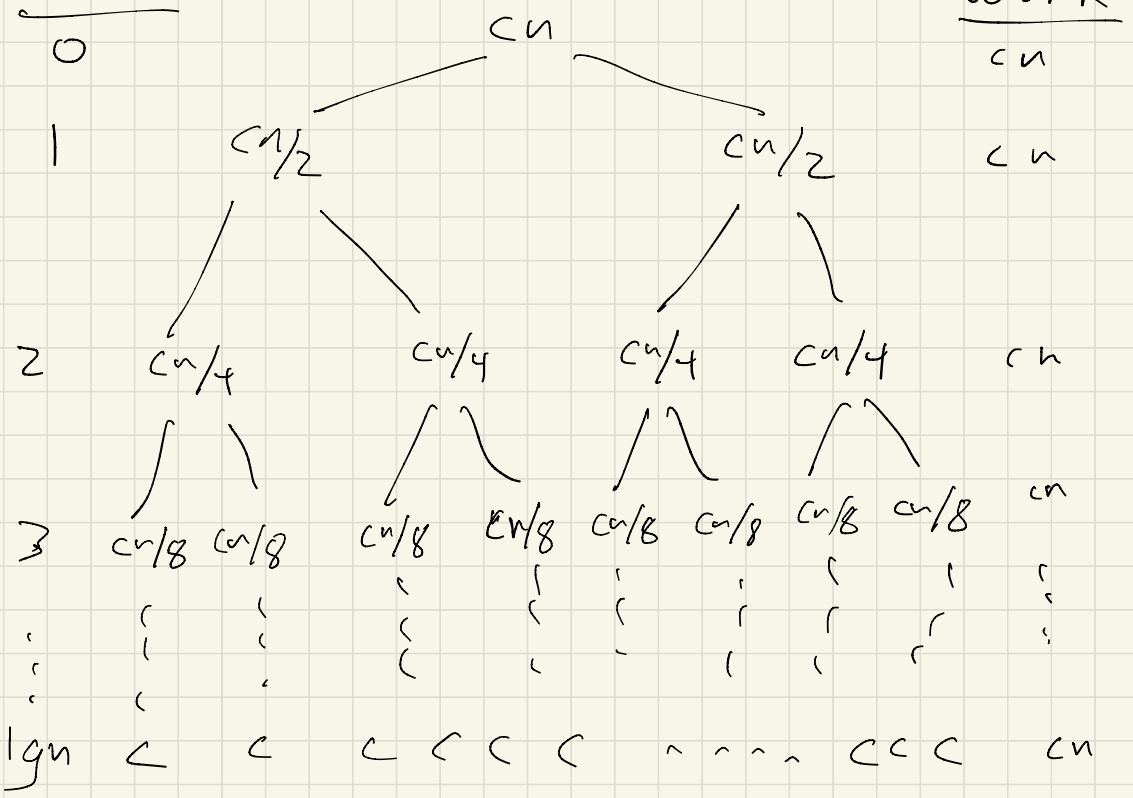


showing 1 level of recursion



show 2 levels of recursion

Showing all levels of recursion



when size is n → level 0 $\quad n/2^0$

" " " $n/2$ → level 1 $\quad n/2 = n/2^1$

" " " $n/4$ → level 2 $\quad n/4 = n/2^2$

⋮

" " " 1 → level l $\quad 1 = n/2^l$

solve for l interms of n:

$2^l = n \rightarrow l = \lg n$

Total work $= \sum_{i=0}^{\lg n} cn = cn(\lg n + 1) = cn\lg n + cn$

$= \Theta(n \lg n)$

Therefore: Running time of Merge-Sort is $\Theta(n \lg n)$.

§ 3.3 Lower bound on comparison-based sorting

Insertion Sort: $\Omega(n)$ $O(n^2)$
Merge Sort: $\Theta(n \lg n)$

Q: Is it possible to do better than $\Theta(n \lg n)$ in the worst-case?

A: No: can not do better than $\Theta(n \lg n)$ for a "comparison-based" sorting alg.

Def: A comparison-based sorting algorithm is an algorithm which can only interface w/ the keys by performing tests of the form "$a_i \le a_j$" "$a_i > a_j$", ... $(\le, <, \ge, =, >)$.

Will show:

Thm: Every comp-based sorting algorithm requires $\Omega(n \lg n)$ comparisons in the worst case.

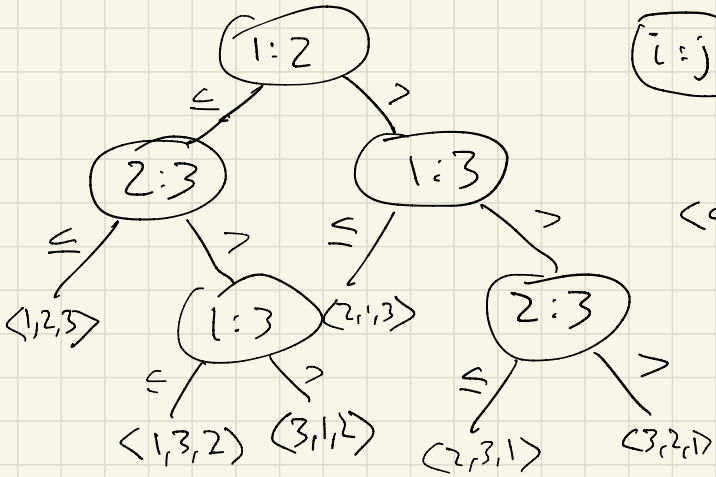
Observation Every comp-based sorting algorithm runs in $\Omega(n)$ time.

Proof: Sorting algorithm must read each of n keys at least once.

To talk about arbitrary sorting algorithm, will use "decision-tree model"

Idea: Can represent all possible paths through algorithm w/ tree. Since can only base decisions on results of comparisons (e.g. "$a_i \le a_j$" either "yes" or "no") tree must be binary.

Decision-Tree for Insertion sort on 3 elements:



(i:j) compare original ith element w/ original jth element

⟨a,b,c⟩ permutation req'd to sort original input.

Example

$\langle 7,9,5 \rangle$

$7 \le 9$?

$9 \le 5$?

$7 \le 5$?

$\langle 3,1,2 \rangle$

$\langle 7,9,5 \rangle$

$\langle 7,9,5 \rangle$

$\langle 7,5,9 \rangle$

$\langle 5,7,9 \rangle$

← perm needed to go from $\langle 7,9,5 \rangle$ to $\langle 5,7,9 \rangle$

Proof of Thm Sps we have some comp-based sorting alg. Sps we have input of size n. Note:

(1) Every possible permutation of n things must occur as leaf on tree.

(2) Thus must have at least $n!$ leaves.

(3) worst case = height of tree $=: h$

(4) Sps ℓ = # leaves, so
$$n! \leq \ell \text{ by (2)}$$
max # of leaves in tree of height h is 2^h in a complete binary tree so
$$\ell \leq 2^h$$

(5) $n! \leq 2^h$, taking lg's

$$\Rightarrow h \geq \lg(n!) = \Omega(n \lg n)$$
(Stirling's)

Thus worst-case # of comparisons needed is $\Omega(n \lg n)$. ▨