"Condition": while we haven't reached end of L and ~~or~~ the end of R

Q. Given L and R of size n, how long does Merge(L, R) take (running time)?

~~(R)~~ $\Theta(n)$ (G) $\Theta(n\log n)$ (B) $\Theta(n^2)$

$O(2n)$ $O(n)$ $\Theta(n)$

So, while loop, in each iteration, increment i_A. The index i_A has max. value $2n-1$. Hence, while loop runs at most (exactly) $2n$ times. The running time is $\Theta(n)$.

$$\boxed{\text{Given L of size } m \text{ and R of size } n\text{, both sorted arrays, merging can be done in } \Theta(n+m) \text{ time.}}$$

MergeSort(A)

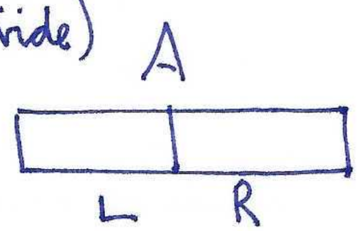
$O(1)$ 1. If size of $A \leqslant 1$, return A. (Base case)

$O(1)$/$O(n)$ 2. $L = A[0..n/2]$, $R = A[\frac{n}{2}+1..n]$ (Divide)

$T(n/2)$ 3. L = MergeSort(L)

$T(n/2)$ 4. R = MergeSort(R)

$O(n)$ 5. Output Merge(L,R) (Conquer)



$T(n)$ = worst case running time of MergeSort on an array of size n

$$T(n) \leq O(1) + O(n) + T(n/2) + T(n/2) + O(n)$$

$$T(n) \leq 2 \cdot T\left(\frac{n}{2}\right) + O(n) \qquad (T(1) = O(1))$$

Recurrence

recursive calls

size of recursion input

Time to merge/combine subsolutions

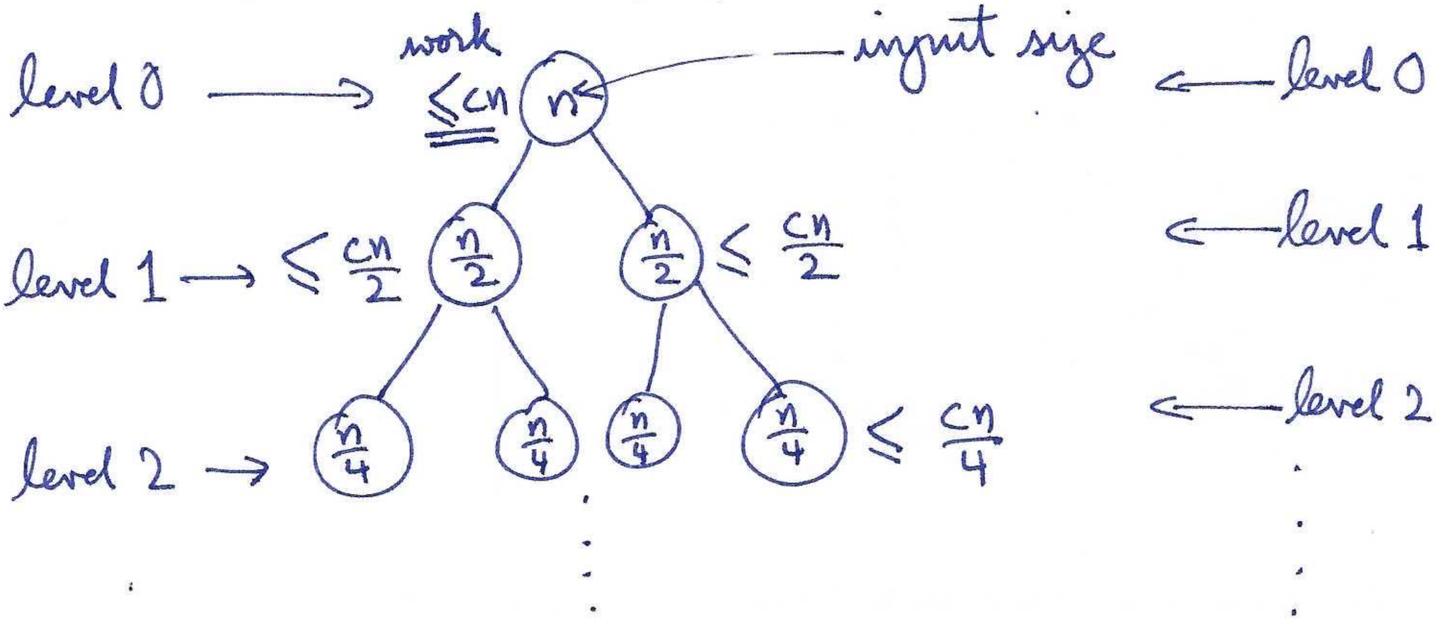
Solve this recurrence

constant

$$T(n) \leq 2T\left(\frac{n}{2}\right) + cn$$

Base case $T(1) \leq c$

Work in a call is running time OTHER than recursive calls



Q. What is input size at level i?

(R) $n/2(i+1)$ ✓(G) $n/2^i$ (B) 2^i

Q. How many calls at level i?

(R) $2(i+1)$ (G) $n/2^i$ ✓(B) 2^i

Work done in a call at level $i \leq \frac{c \cdot n}{2^i}$

Q. What is the total work done in ALL calls at level i?

✓(R) $\leq cn$ (G) $\leq cn \lg n$ (B) $\leq cn \times 2^i$

Total work in ALL calls at level i

$\leq$ (# calls at level i) $\times$ (max work in any call at level i)

$$\leq \cancel{2^i} \times \frac{cn}{\cancel{2^i}} = cn$$

$\#\text{levels} \leq \lceil \lg_2 n \rceil$

Total work overall = Total running time

$$= \sum_{i=0}^{\lg_2 n} (\text{Total work in all calls at level } i)$$

$$\leq \sum_{i=0}^{\lceil \lg_2 n \rceil} cn \leq cn \lceil \lg_2 n \rceil \boxed{= O(n \log n)}$$

Punchlines:

(1) Merge runs in linear time.
sorted arrays

(2) Mergesort runs in $\Theta(n\log n)$.
$O(n\log n)$

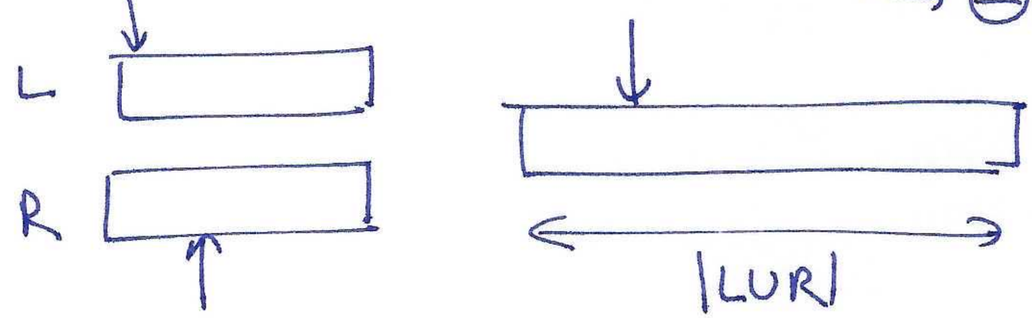
(3) ANY algorithm with the recurrence

$$T(n) \leq 2T\left(\frac{n}{2}\right) + cn \qquad T(1) \leq c$$

runs in $T(n) = O(n\log n)$

Quicksort

Merging requires extra memory. $\Theta(n)$



NOT "in-place"

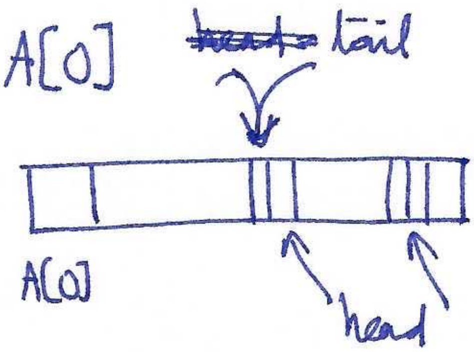
In-place sorting: requires $O(\log n)$ extra memory

Heapsort

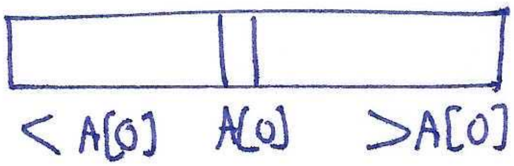
Quicksort (A) (n = length of A)

1) If $n=1$, do nothing and return.

2) Partition $A[1..n-1]$ using pivot $= A[0]$



3) Swap $A[0]$ with $A[tail]$ (and you will get)



sum of sizes $\leq n-1$

4) Quicksort $(A[0..tail-1])$

5) Quicksort ~~(A[0..head~~ $(A[head..n-1])$

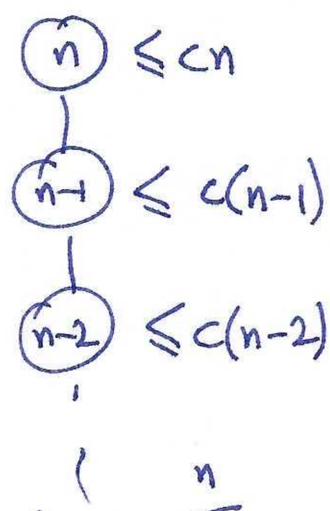
What is the worst-case recurrence for Quicksort?

~~(R)~~ $T(n) \leq 2T(n/2) + O(n)$ tail $> \frac{n}{2}$

(G) $T(n) \leq 2T(n-1) + O(n)$

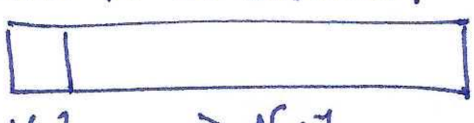
✓ ~~(B)~~ $T(n) \leq T(n-1) + O(n)$

$\geq$



In worst-case, Quicksort takes $\Omega(n^2)$ time.

When A is sorted!

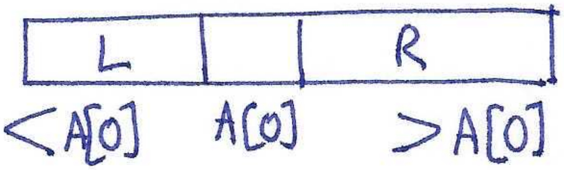


Total work $\leq c\sum_{i=0}^{n} n$

$= O(n^2)$

1) Pick a "pivot" (A[0])

2) <u>Partition</u> A by pivot

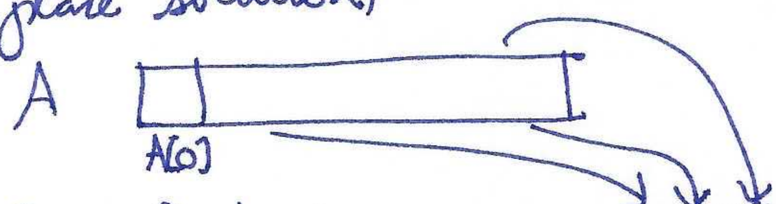


(Assume elements are distinct)

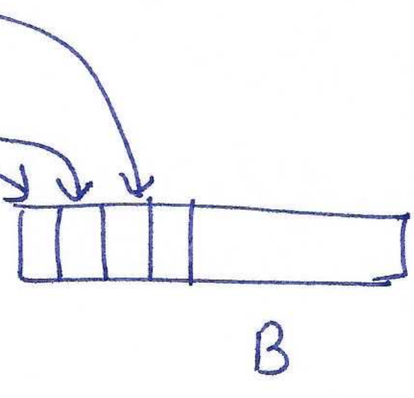
3) Recursively sort <A[0] part and >A[0] part

How to partition

Simple (not in-place solution)



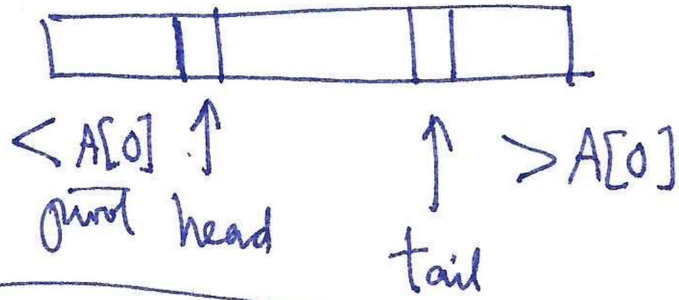
0) Create empty array B of ~~size~~ same size



1) Single for loop over A to copy every element <A[0] into B

2) Insert A[0] into B (at the end of copied elements)

3) Another for loop over A to copy every element >A[0] into B.



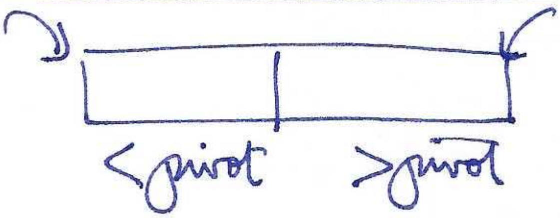
If A[head] < A[0], head++

If A[tail] > A[0], tail--

When both stop, A[head] > A[0] and A[tail] < A[0].

Swap A[head] and A[tail], and continue.

Partition(A, pivot)



(Assuming values are distinct)

1. head = 0, tail = A.length − 1

2. while (head < tail)

   (a) while (A[head] < pivot), head++

   (b) while (A[tail] > pivot), tail--

   (c) swap A[head] with A[tail]

Partition runs in $\Theta(n)$ time.

Randomized Quicksort

Pick pivot at random.

Pick uniform random i and swap $A[0]$ with $A[i]$.

Run Quicksort

$O(n \log n)$ averaged over randomness

All sorting algorithms discussed run in $\Theta(n \log n)$.

We cannot beat this bound!

(unless additional assumptions on the input are made)

No sorting algorithm that only uses comparisons can run in time better than $n \ln n$.