MergeSort

Goal: sort a given list/array of elements

\[9, 3, 6, 2, 1, 7, 8, 0\]

MergeSort - a divide-and-conquer algorithm

1) divide - split the input into a few parts (typically 2)
2) solve each part (recursively)
3) conquer - put the solutions of the parts together to get the solution for the original input

After 1) I have

\[9, 3, 6, 2\]

\[1, 7, 8, 0\]

After reunion

\[2, 3, 6, 9\]

\[0, 1, 7, 8\]

In step 3):

\[0, 1, 2, 3, 6, 7, 8, 9\]

Result
**MergeSort**

How to “conquer”?  
A = 2, 3, 0, 1  
B = 8, 9  
C = 0, 1, 2, 3, 5, 6, 7, 8, 9  

for merge  

**Pseudocode:**

Jack’s suggestion:  
- compare the first elements \( A[0] \) and \( B[0] \)  
  
```
merge ( array A , array B )
```
  
i = 0  
j = 0  
k = 0  
A[ A.length ] = maxint  
B[ B.length ] = maxint  

While \( k < A.length + B.length \)  
   
if \( A[ i ] < B[ j ] \) then  
   
   \( C[ k ] = A[ i ] \); \( i++ \); \( k++ \)  
else  
   
   \( C[ k ] = B[ j ] \); \( j++ \); \( k++ \)  

}  

return C  

Mike’s suggestion:  
- use binary search to insert elements from \( B \) into \( A \)  
- we spend \( O(\log n) \) time per element from \( B \) to find its new position  
- but we might have problems with inserting  
- even if we do insert in \( O(1) \) time per element,  
- we do the “merge” in time \( O(n \log n) \) plus  
- we need to consider the recursive calls  
- and the overall time will be more than \( \Omega(n \log n) \)

**Running time:**  
\( O( A.length + B.length ) \) in our case:  
\( O(n) \)

MERGE works just fine on sorted linked lists
MergeSort pseudocode:

```
MERGE_SORT (an array X):
  0. if X.length = 1, then return X
  1. create A with elements X[0, ..., \( \frac{X.length}{2} \)]
  2. create B with elements X[\( \frac{X.length}{2} \), ..., X.length]
  3. A = MERGE_SORT(A)
  4. B = MERGE_SORT(B)
  5. RETURN MERGE(A, B)
```

Running time:

- Input: \( \leq cn \) steps
- Build array: \( \leq 2 \cdot (c \cdot \frac{n}{2}) \) steps
- Build array: \( \leq 4 \cdot (c \cdot \frac{n}{4}) \) steps
- \( \vdots \)
- How many levels: \( O(\log n) \)
- Steps 1, 2, 5: \( O(n) \) time \( \leq cn \) steps total
- On level \( k \): we spend \( 2^k \cdot (c \cdot \frac{n}{2^k}) \) steps in step 1, 2, 5 in all recursive calls on this level
- Since \( 2^k \cdot (c \cdot \frac{n}{2^k}) = cn \)

TOTAL: \( O(\log n) \) levels, \( cn \) steps per level

\( = \boxed{O(n \log n)} \)
MergeSort

A word about recurrences:

\[
\begin{aligned}
T(n) &\leq 2T\left(\frac{n}{2}\right) + cn \\
&\leq 2 \left(2T\left(\frac{n}{4}\right) + c \frac{n}{4}\right) + cn = 4T\left(\frac{n}{4}\right) + cn + cn \\
&\leq 4 \left(2T\left(\frac{n}{8}\right) + c \frac{n}{8}\right) + 2cn = 8T\left(\frac{n}{8}\right) + 3cn \leq \ldots \leq 2^k T\left(\frac{n}{2^k}\right) + kcn
\end{aligned}
\]

we stop with \( T(1) \), i.e. when \( \frac{n}{2^k} = 1 \Rightarrow k = \log n \)

then \( 2^k T\left(\frac{n}{2^k}\right) + kcn = cn + cn \log n = O(n \log n) \)
QuickSort

Another divide-and-conquer sorting algorithm:

A = 7 5 2 1 8 4 24

rearranged:

2 1 4 7 5 8 24

QUICKSORT (array A)
0) if A.length ≤ 4 then exit
1) choose a pivot element (let it be x)
2) rearrange A so that
a) the elements < x come before x
b) the elements > x come after x
c) [not every literature specifies] the elements = x are next to x
3) let j be the position of x in the rearranged A
4) QUICKSORT (A[0...j-1])
5) QUICKSORT (A[j+1...A.length-1])
A detailed pseudocode (using left-right marks to denote the part of the array we are working on)

**QUICKSORT** (Input: an array A, indices left & right)

0. if left ≥ right then exit

1. let $z$ be a random number from {left, left+1, ..., right}  
   → we are sorting A from left to right

1.5. let pivot = $A[z]$  
   $N$x on previous slide

2. REARRANGE (A, left, right, pivot)

3. for (j = left; (j < right) v (A[j] ≠ pivot); j++)  
   // you are welcome to do this in a while loop
   // j = position of the pivot in the rearranged array

4. QUICKSORT (A, left, j-1)

5. QUICKSORT (A, j+1, right)

(ALMOST) DONE ☺

if we are lucky and pivot is in the middle, the recursive calls take $T(n/2)$ each

**IN THE LUCKY CASE:** $T(n) \leq T(n/2) + cn$  
just like MERGESORT

$\Rightarrow O(n \log n)$ time
QuickSort

REARRANGE (Input: an array A, indices left and right, pivot)

// version 1 with 2 arrays
0. \( LB = left; \ rB = right; \)
1. for (i = left; i ≤ right; i++) {
2.     if (A[i] < pivot) then
4.     else if (A[i] > pivot) then
5.         B[rB] = A[i]; rB--;
6. }
7. for (i = LB; i ≤ rB; i++) B[i] = pivot;
8. for (i = left; i ≤ right; i++) A[i] = B[i]

\( \O(n) \)

\( LB = rB \)

where \( n = \) # elements in A from left to right

\( = right - left + 1 \)

not in-place \( \equiv \) because we are using extra space (the B array)

0(n)

See the book for how to do the swapping in place

\( \leq \text{pivot} \)

\( \geq \text{pivot} \)
QuickSort

Running time:

if we are lucky:  $O(n \log n)$

worst-case: if we happen to pick the largest (or the smallest) element as pivot

→ then recursive calls on $n-1$ elements and $0$ elements

we would spend

$$cn + c(n-1) + c(n-2) + \ldots + c = c \frac{n+1}{2} \cdot n = O(n^2)$$

expected run.time:  $O(n \log n)$ with high probability
QuickSort

A note about in-place sorting:
Thm: Every comparison-based sort needs $\Omega(n \log n)$ steps.
BucketSort and RadixSort

BucketSort (also known as counting sort)

Note: BucketSort is a stable sort.
BucketSort and RadixSort

RadixSort:

Running time:
Comparing various sorts

- Why do we have so many sorting algorithms?